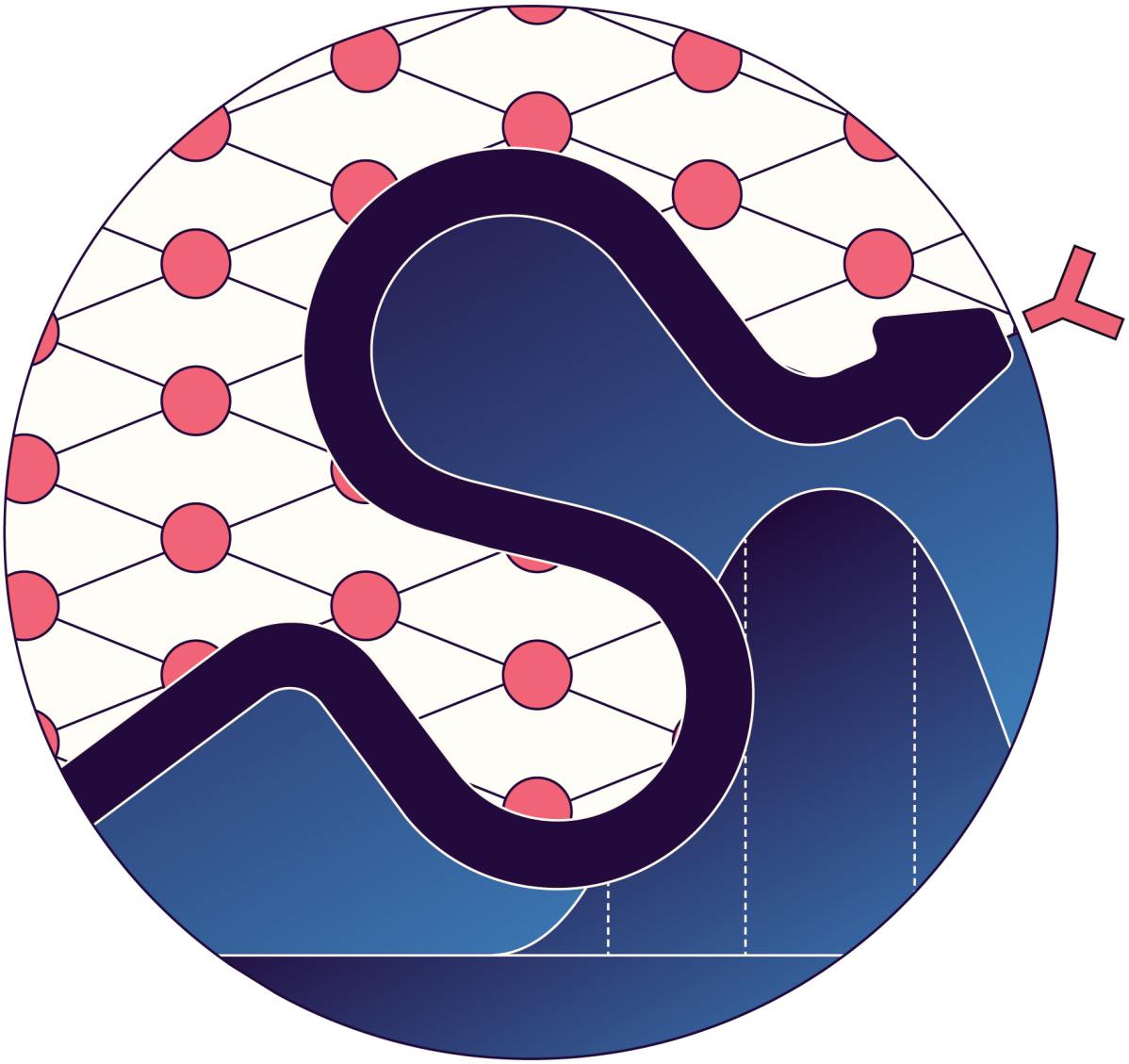


Proceedings of the 21st Python in Science Conference

PROCEEDINGS OF THE 21ST PYTHON IN SCIENCE CONFERENCE

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

SciPy 2022
Austin, Texas
July 11 - July 17, 2022

ISSN:2575-9752
https://doi.org/10.25080/majora-212e5952-046

ORGANIZATION

Conference Chairs

JONATHAN GUYER, NIST
ALEXANDRE CHABOT-LECLERC, Enthought, Inc.

Program Chairs

MATT HABERLAND, Cal Poly
JULIE HOLLEK, Mozilla
MADICKEN MUNK, University of Illinois
GUEN PRAWIROATMODJO, Microsoft Corp

Communications

ARLISS COLLINS, NumFOCUS
MATT DAVIS, Populus
DAVID NICHOLSON, Embedded Intelligence

Birds of a Feather

ANDREW REID, NIST
ANASTASIIA SARMAKEEVA, George Washington University

Proceedings

MEGHANN AGARWAL, Overhaul
CHRIS CALLOWAY, University of North Carolina
DILLON NIEDERHUT, Novi Labs
DAVID SHUPE, Caltech's IPAC Astronomy Data Center

Financial Aid

SCOTT COLLIS, Argonne National Laboratory
NADIA TAHIRI, Université de Montréal

Tutorials

MIKE HEARNE, USGS
LOGAN THOMAS, Enthought, Inc.

Sprints

TANIA ALLARD, Quansight Labs
BRIGITTA SIPŐCZ, Caltech/IPAC

Diversity

CELIA CINTAS, IBM Research Africa
BONNY P MCCLAIN, O'Reilly Media
FATMA TARLACI, OpenTeams

Activities

PAUL ANZEL, Codecov
INESSA PAWSON, Albus Code

Sponsors

KRISTEN LEISER, Enthought, Inc.

Financial

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

Logistics

KRISTEN LEISER, Enthought, Inc.

Proceedings Reviewers

Aileen Nielsen
Ajit Dhobale
Alejandro Coca-Castro
Alexander Yang
Bhupendra A Raut
Bradley Dice
Brian Gue
Cadiou Corentin
Carl Simon Adorf
Chen Zhang
Chiara Marmo
Chitaranjan Mahapatra
Chris Calloway
Daniel Wheeler
David Nicholson
David Shupe
Dillon Niederhut
Diptorup Deb
Jelena Milosevic
Michal Maciejewski
Ed Rogers
Himaghna Bhattacharjee
Hongsup Shin
Indraneil Paul
Ivan Marroquin
James Lamb
Jyh-Miin Lin
Jyotika Singh
Karthik Murugadoss
Kehinde Ajayi
Kelly L. Rowland
Kelvin Lee
Kevin Maik Jablonka
Kevin W. Beam
Kuntao Zhao
Maruthi NH
Matt Craig
Matthew Feickert
Meghann Agarwal
Melissa Weber Mendonça
Onuralp Soylemez
Rohit Goswami
Ryan Bunney
Shubham Sharma
Siddhartha Srivastava
Sushant More
Tetsuo Koyama
Thomas Nicholas
Victoria Adesoba
Vidhi Chugh
Vivek Sinha
Wenduo Zhou
Zuhal Cakir

Accepted Talk Slides

Building Binary Extensions with pybind11, scikit-build, and cibuildwheel, Henry Schreiner, and Joe Rickerby, and Ralf Grosse-Kunstleve, and Wenzel Jakob, and Matthieu Darbois, and Aaron Gokaslan, and Jean-Christophe Fillion-Robin, and Matt McCormick
doi.org/10.25080/majora-212e5952-033
Python Development Schemes for Monte Carlo Neutronics on High Performance Computing, Jackson P. Morgan, and Kyle E. Niemeyer
doi.org/10.25080/majora-212e5952-034
Awkward Packaging: Building scikit-HEP, Henry Schreiner, and Jim Pivarski, and Eduardo Rodrigues
doi.org/10.25080/majora-212e5952-035
Development of Accessible, Aesthetically-Pleasing Color Sequences, Matthew A. Petroff
doi.org/10.25080/majora-212e5952-036
Cutting Edge Climate Science in the Cloud with Pangeo, Julius Busecke
doi.org/10.25080/majora-212e5952-037
Pylira: deconvolution of images in the presence of Poisson noise, Axel Donath, and Aneta Siemiginowska, and Vinay Kashyap, and Douglas Burke, and Karthik Reddy Solipuram, and David van Dyk
doi.org/10.25080/majora-212e5952-038
Accelerating Science with the Generative Toolkit for Scientific Discovery (GT4SD), GT4SD team
doi.org/10.25080/majora-212e5952-039
MModel: a modular modeling framework for scientific prototyping, Peter Sun, and John A. Marohn
doi.org/10.25080/majora-212e5952-03a
Monaco: Quantify Uncertainty and Sensitivities in Your Computational Models with a Monte Carlo Library, W. Scott Shambaugh
doi.org/10.25080/majora-212e5952-03b
UFuncs and DTypes: new possibilities in NumPy, Sebastian Berg, and Stéfan van der Walt
doi.org/10.25080/majora-212e5952-03c
Per Python ad astra: interactive Astrodynamics with poliastro, Juan Luis Cano Rodríguez
doi.org/10.25080/majora-212e5952-03d
pyampute: a Python library for data amputation, Rianne M Schouten, and Davina Zamanzadeh, and Prabhant Singh
doi.org/10.25080/majora-212e5952-03e
Scientific Python: From GitHub to TikTok, Juanita Gomez Romero, and Stéfan van der Walt, and K. Jarrod Millman, and Melissa Weber Mendonça, and Inessa Pawson
doi.org/10.25080/majora-212e5952-03f
Scientific Python: By maintainers, for maintainers, Pamphile T. Roy, and Stéfan van der Walt, and K. Jarrod Millman, and Melissa Weber Mendonça
doi.org/10.25080/majora-212e5952-040
Improving random sampling in Python: scipy.stats.sampling and scipy.stats.qmc, Pamphile T. Roy, and Matt Haberland, and Christoph Baumgarten, and Tirth Patel
doi.org/10.25080/majora-212e5952-041
Petabyte-scale ocean data analytics on staggered grids via the grid ufunc protocol in xGCM, Thomas Nicholas, and Julius Busecke, and Ryan Abernathey
doi.org/10.25080/majora-212e5952-042

Accepted Posters

Optimal Review Assignments for the SciPy Conference Using Binary Integer Linear Programming in SciPy 1.9, Matt Haberland, and Nicholas McKibben
doi.org/10.25080/majora-212e5952-029
Contributing to Open Source Software: From not knowing Python to becoming a Spyder core developer, Daniel Althviz Moré
doi.org/10.25080/majora-212e5952-02a
Semi-Supervised Semantic Annotator (S3A): Toward Efficient Semantic Image Labeling, Nathan Jessurun, and Olivia P. Dizon-Paradis, and Dan E. Capecci, and Damon L. Woodard, and Navid Asadizanjani
doi.org/10.25080/majora-212e5952-02b
Bioframe: Operating on Genomic Interval Dataframes, Nezar Abdennur, and Geoffrey Fudenberg, and Ilya M. Flyamer, and Aleksandra Galitsyna, and Anton Goloborodko, and Maxim Imakaev, and Trevor Manz, and Sergey V. Venev
doi.org/10.25080/majora-212e5952-02c
Likeness: a toolkit for connecting the social fabric of place to human dynamics, Joseph V. Tuccillo, and James D. Gaboardi
doi.org/10.25080/majora-212e5952-02d
pyAudioProcessing: Audio Processing, Feature Extraction, and Machine Learning Modeling, Jyotika Singh
doi.org/10.25080/majora-212e5952-02e
Kiwi: Python Tool for Tex Processing and Classification, Neelima Pulagam, and Sai Marasani, and Brian Sass
doi.org/10.25080/majora-212e5952-02f

PHYLOGEOGRAPHY: ANALYSIS OF GENETIC AND CLIMATIC DATA OF SARS-COV-2, Wanlin Li, and Aleksandr Koshkarov, and My-Linh Luu, and Nadia Tahiri
doi.org/10.25080/majora-212e5952-030
DESIGN OF A SCIENTIFIC DATA ANALYSIS SUPPORT PLATFORM, Nathan Martindale, and Jason Hite, and Scott Stewart, and Mark Adams
doi.org/10.25080/majora-212e5952-031
OPENING ARM: A PIVOT TO COMMUNITY SOFTWARE TO MEET THE NEEDS OF USERS AND STAKEHOLDERS OF THE PLANET'S LARGEST CLOUD OBSERVATORY, Zachary Sherman, and Scott Collis, and Max Grover, and Robert Jackson, and Adam Theisen
doi.org/10.25080/majora-212e5952-032

SCIPY TOOLS PLENARIES

SCIPY TOOLS PLENARY - CEL TEAM, Inessa Pawson
doi.org/10.25080/majora-212e5952-043
SCIPY TOOLS PLENARY ON MATPLOTLIB, Elliott Sales de Andrade
doi.org/10.25080/majora-212e5952-044
SCIPY TOOLS PLENARY - NUMPY, Inessa Pawson
doi.org/10.25080/majora-212e5952-045

LIGHTNING TALKS

DOWNSAMPLING TIME SERIES DATA FOR VISUALIZATIONS, Delaina Moore
doi.org/10.25080/majora-212e5952-027
ANALYSIS AS APPLICATIONS: QUICK INTRODUCTION TO LOCKFILES, Matthew Feickert
doi.org/10.25080/majora-212e5952-028

SCHOLARSHIP RECIPIENTS

AMAN GOEL, University of Delhi
ANURAG SAHA ROY, Saarland University
ISURU FERNANDO, University of Illinois at Urbana Champaign
KELLY MEEHAN, US Forest Service
KADAMBARI DEVARAJAN, University of Rhode Island
KRISHNA KATYAL, Thapar Institute of Engineering and Technology
MATTHEW MURRAY, Dask
NAMAN GERA, Sympy, LPython
ROHIT GOSWAMI, University of Iceland
SIMON CROSS, QuTIP
TANYA AKUMU, IBM Research
ZUHAL CAKIR, Purdue University

CONTENTS

The Advanced Scientific Data Format (ASDF): An Update

Perry Greenfield[‡*], Edward Slavich[‡†], William Jamieson[‡†], Nadia Dencheva[‡†]

Abstract—We report on progress in developing and extending the new (ASDF) format we have developed for the data from the James Webb and Nancy Grace Roman Space Telescopes since we reported on it at a previous Scipy. While the format was developed as a replacement for the long-standard FITS format used in astronomy, it is quite generic and not restricted to use with astronomical data. We will briefly review the format, and extensions and changes made to the standard itself, as well as to the reference Python implementation we have developed to support it. The standard itself has been clarified in a number of respects. Recent improvements to the Python implementation include an improved framework for conversion between complex Python objects and ASDF, better control of the configuration of extensions supported and versioning of extensions, tools for display and searching of the structured metadata, better developer documentation, tutorials, and a more maintainable and flexible schema system. This has included a reorganization of the components to make the standard free from astronomical assumptions. A important motivator for the format was the ability to support serializing functional transforms in multiple dimensions as well as expressions built out of such transforms, which has now been implemented. More generalized compression schemes are now enabled. We are currently working on adding chunking support and will discuss our plan for further enhancements.

Index Terms—data formats, standards, world coordinate systems, yaml

Introduction

The Advanced Scientific Data Format (ASDF) was originally developed in 2015. That original version was described in a paper [Gre15]. That paper described the shortcomings of the widely used astronomical standard format FITS [FIT16] as well as those of existing potential alternatives. It is not the goal of this paper to rehash those points in detail, though it is useful to summarize the basic points here. The remainder of this paper will describe where we are using ASDF, what lessons we have learned from using ASDF for the James Webb Space Telescope, and summarize the most important changes we have made to the standard, the Python library that we use to read and write ASDF files, and best practices for using the format.

We will give an example of a more advanced use case that illustrates some of the powerful advantages of ASDF, and that its application is not limited to astronomy, but suitable for much of scientific and engineering data, as well as models. We finish by outlining our near term plans for further improvements and extensions.

* *Corresponding author: perry@stsci.edu*
‡ *Space Telescope Science Institute*
† *These authors contributed equally.*

Summary of Motivations

- Suitable as an archival format:
  - Old versions continue to be supported by libraries.
  - Format is sufficiently transparent (e.g., not requiring extensive documentation to decode) for the fundamental set of capabilities.
  - Metadata is easily viewed with any text editor.
- Intrinsically hierarchical
- Avoids duplication of shared items
- Based on existing standard(s) for metadata and structure
- No tight constraints on attribute lengths or their values.
- Clearly versioned
- Supports schemas for validating files for basic structure and value requirements
- Easily extensible, both for the standard, and for local or domain-specific conventions.

Basics of ASDF Format

- Format consists of a YAML header optionally followed by one or more binary blocks for containing binary data.
- The YAML [http://yaml.org] header contains all the metadata and defines the structural relationship of all the data elements.
- YAML tags are used to indicate to libraries the semantics of subsections of the YAML header that libraries can use to construct special software objects. For example, a tag for a data array would indicate to a Python library to convert it into a numpy array.
- YAML anchors and alias are used to share common elements to avoid duplication.
- JSON Schema [http://json-schema.org/specification.html], [http://json-schema.org/understanding-json-schema/] is used for schemas to define expectations for tag content and whole headers combined with tools to validate actual ASDF files against these schemas.
- Binary blocks are referenced in the YAML to link binary data to YAML attributes.
- Support for arrays embedded in YAML or in a binary block.

- Streaming support for a single binary block.
- Permit local definitions of tags and schemas outside of the standard.
- While developed for astronomy, useful for general scientific or engineering use.
- Aims to be language neutral.

Current and planned uses

James Webb Space Telescope (JWST)

NASA requires JWST data products be made available in the FITS format. Nevertheless, all the calibration pipelines operate on the data using an internal objects very close to the the ASDF representation. The JWST calibration pipeline uses ASDF to serialize data that cannot be easily represented in FITS, such as World Coordinate System information. The calibration software is also capable of reading and producing data products as pure ASDF files.

Nancy Grace Roman Space Telescope

This telescope, with the same mirror size as the Hubble Space Telescope (HST), but a much larger field of view than HST, will be launched in 2026 or thereabouts. It is to be used mostly in survey mode and is capable of producing very large mosaicked images. It will use ASDF as its primary data format.

Daniel K Inoue Solar Telescope

This telescope is using ASDF for much of the early data products to hold the metadata for a combined set of data which can involve many thousands of files. Furthermore, the World Coordinate System information is stored using ASDF for all the referenced data.

Vera Rubin Telescope (for World Coordinate System interchange)

There have been users outside of astronomy using ASDF, as well as contributors to the source code.

Changes to the standard (completed and proposed)

These are based on lessons learned from usage.

The current version of the standard is 1.5.0 (1.6.0 being developed).

The following items reflect areas where we felt improvements were needed.

Changes for 1.5

Moving the URI authority from `stsci.edu` to `asdf-format.org`

This is to remove the standard from close association with STScI and make it clear that the format is not intended to be controlled by one institution.

Moving astronomy-specific schemas out of standard

These primarily affect the previous inclusion of World Coordinate Tags, which are strongly associated with astronomy. Remaining are those related to time and unit standards, both of obvious generality, but the implementation must be based on some standards, and currently the astropy-based ones are as good or better than any.

Changes for 1.6

Addition of the manifest mechanism

The manifest is a YAML document that explicitly lists the tags and other features introduced by an extension to the ASDF standard. It provides a more straightforward way of associating tags with schemas, allowing multiple tags to share the same schema, and generally making it simpler to visualize how tags and schemas are associated (previously these associations were implied by the Python implementation but were not documented elsewhere).

Handling of null values and their interpretation

The standard didn't previously specify the behavior regarding null values. The Python library previously removed attributes from the YAML tree when the corresponding Python attribute has a `None` value upon writing to an ADSF file. On reading files where the attribute was missing but the schema indicated a default value, the library would create the Python attribute with the default. As mentioned in the next item, we no longer use this mechanism, and now when written, the attribute appears in the YAML tree with a null value if the Python value is `None` and the schema permits null values.

Interpretation of default values in schema

The use of default values in schemas is discouraged since the interpretation by libraries is prone to confusion if the assemblage of schemas conflict with regard to the default. We have stopped using defaults in the Python library and recommend that the ASDF file always be explicit about the value rather than imply it through the schema. If there are practical cases that preclude always writing out all values (e.g., they are only relevant to one mode and usually are irrelevant), it should be the library that manages whether such attributes are written conditionally rather using the schema default mechanism.

Add alternative tag URI scheme

We now recommend that tag URIs begin with `asdf://`

Be explicit about what kind of complex YAML keys are supported

For example, not all legal YAML keys are supported. Namely YAML arrays, which are not hashable in Python. Likewise, general YAML objects are not either. The Standard now limits keys to string, integer, or boolean types. If more complex keys are required, they should be encoded in strings.

Still to be done

Upgrade to JSON Schema draft-07

There is interest in some of the new features of this version, however, this is problematic since there are aspects of this version that are incompatible with *draft-04*, thus requiring all previous schemas to be updated.

Replace extensions section of file history

This section is considered too specific to the concept of Python extensions, and is probably best replaced with a more flexible system for listing extensions used.

Changes to Python ASDF package

Easier and more flexible mechanism to create new extensions (2.8.0)

The previous system for defining extensions to ASDF, now deprecated, has been replaced by a new system that makes the association between tags, schemas, and conversion code more straightforward, as well as providing more intuitive names for the methods and attributes, and makes it easier to handle reference cycles if they are present in the code (also added to the original Tag handling classes).

Introduced global configuration mechanism (2.8.0)

This reworks how ASDF resources are located, and makes it easier to update the current configuration, as well as track down the location of the needed resources (e.g., schemas and converters), as well as removing performance issues that previously required extracting information from all the resource files thus slowing the first `asdf.open` call.

Added `info/search` methods and command line tools (2.6.0)

These allow displaying the hierarchical structure of the header and the values and types of the attributes. Initially, such introspection stopped at any tagged item. A subsequent change provides mechanisms to see into tagged items (next item). An example of these tools is shown in a later section.

Added mechanism for info to display tagged item contents (2.9.0)

This allows the library that converts the YAML to Python objects to expose a summary of the contents of the object by supplying an optional "dunder" method that the info mechanism can take advantage of.

Added documentation on how ASDF library internals work

These appear in the readthedocs under the heading "Developer Overview".

Plugin API for block compressors (2.8.0)

This enables a localized extension to support further compression options.

Support for `asdf://` URI scheme (2.8.0)

Support for ASDF Standard 1.6.0 (2.8.0)

This is still subject to modifications to the 1.6.0 standard.

Modified handling of defaults in schemas and `None` values (2.8.0)

As described previously.

Using ASDF to store models

This section highlights one aspect of ASDF that few other formats support in an archival way, e.g., not using a language-specific mechanism, such as Python's pickle. The astropy package contains a modeling subpackage that defines a number of analytical, as well as a few table-based, models that can be combined in many ways, such as arithmetically, in composition, or multi-dimensional. Thus it is possible to define fairly complex multi-dimensional models, many of which can use the built in fitting machinery.

These models, and their compound constructs can be saved in ASDF files and later read in to recreate the corresponding astropy objects that were used to create the entries in the ASDF file. This is made possible by the fact that expressions of models are straightforward to represent in YAML structure.

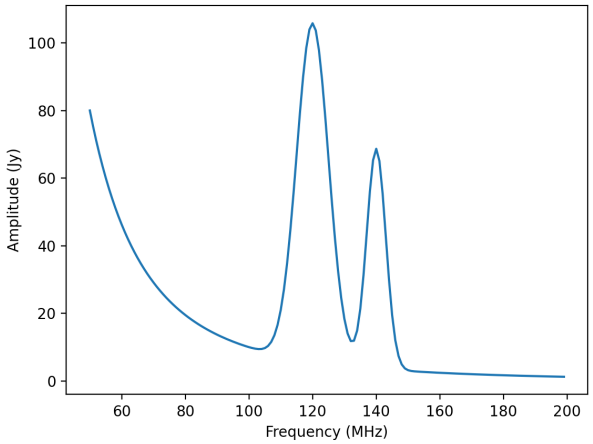


Fig. 1: *A plot of the compound model defined in the first segment of code.*

Despite the fact that the models are in some sense executable, they are perfectly safe so long as the library they are implemented in is safe (e.g., it doesn't implement an "execute any OS command" model). Furthermore, the representation in ASDF does not explicitly use Python code. In principle it could be written or read in any computer language.

The following illustrates a relatively simple but not trivial example.

First we define a 1D model and plot it.

```
import numpy as np
import astropy.modeling.models as amm
import astropy.units as u
import asdf
from matplotlib import pyplot as plt

# Define 3 model components with units
g1 = amm.Gaussian1D(amplitude=100*u.Jy,
                    mean=120*u.MHz,
                    stddev=5.*u.MHz)
g2 = amm.Gaussian1D(65*u.Jy, 140*u.MHz, 3*u.MHz)
powerlaw = amm.PowerLaw1D(amplitude=10*u.Jy,
                          x_0=100*u.MHz,
                          alpha=3)
# Define a compound model
model = g1 + g2 + powerlaw
x = np.arange(50, 200) * u.MHz
plt.plot(x, model(x))
```

The following code will save the model to an ASDF file, and read it back in

```
af = asdf.AsdfFile()
af.tree = {'model': model}
af.write_to('model.asdf')
af2 = asdf.open('model.asdf')
model2 = af2['model']
model2 is model
    False
model2(103.5) == model(103.5)
    True
```

Listing the relevant part of the ASDF file illustrates how the model has been saved in the YAML header (reformatted to fit in this paper column).

```
model: !transform/add-1.2.0
  forward:
```

```
  - !transform/add-1.2.0
    forward:
    - !transform/gaussian1d-1.0.0
      amplitude: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 Jy, value: 100.0}
      bounding_box:
      - !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 92.5}
      - !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 147.5}
      bounds:
        stddev: [1.1754943508222875e-38, null]
      inputs: [x]
      mean: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 120.0}
      outputs: [y]
      stddev: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 5.0}
    - !transform/gaussian1d-1.0.0
      amplitude: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 Jy, value: 65.0}
      bounding_box:
      - !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 123.5}
      - !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 156.5}
      bounds:
        stddev: [1.1754943508222875e-38, null]
      inputs: [x]
      mean: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 140.0}
      outputs: [y]
      stddev: !unit/quantity-1.1.0
        {unit: !unit/unit-1.0.0 MHz, value: 3.0}
    inputs: [x]
    outputs: [y]
  - !transform/power_law1d-1.0.0
    alpha: 3.0
    amplitude: !unit/quantity-1.1.0
      {unit: !unit/unit-1.0.0 Jy, value: 10.0}
    inputs: [x]
    outputs: [y]
    x_0: !unit/quantity-1.1.0
      {unit: !unit/unit-1.0.0 MHz, value: 100.0}
  inputs: [x]
  outputs: [y]
...
```

Note that there are extra pieces of information that define the model more precisely. These include:

- many tags indicating special items. These include different kinds of transforms (i.e., functions), quantities (i.e., numbers with units), units, etc.
- definitions of the units used.
- indications of the valid range of the inputs or parameters (bounds)
- each function shows the mapping of the inputs and the naming of the outputs of each function.
- the addition operator is itself a transform.

Without the use of units, the YAML would be simpler. But the point is that the YAML easily accommodates expression trees. The tags are used by the library to construct the astropy models, units and quantities as Python objects. However, nothing in the above requires the library to be written in Python.

This machinery can handle multidimensional models and supports both the combining of models with arithmetic operators as well as pipelining the output of one model into another. This system has been used to define complex coordinate transforms from telescope detectors to sky coordinates for imaging, and wavelengths for spectrographs, using over 100 model components, something that the FITS format had no hope of managing, nor any other scientific format that we are aware of.

Displaying the contents of ASDF files

Functionality has been added to display the structure and content of the header (including data item properties), with a number of options of what depth to display, how many lines to display, etc. An example of the info use is shown in Figure 2.

There is also functionality to search for items in the file by attribute name and/or values, also using pattern matching for either. The search results are shown as attribute paths to the items that were found.

ASDF Extension/Converter System

There are a number of components that are involved. Converters encapsulate the code that handles converting Python objects to and from their ASDF representation. These are classes that inherit from the basic `Converter` class and define two Class attributes: tags, types each of which is a list of associated tag(s) and class(es) that the specific converter class will handle (each converter can handle more than one tag type and more than one class). The ASDF machinery uses this information to map tags to converters when reading ASDF content, and to map types to converters when saving these objects to an ASDF file.

Each converter class is expected to supply two methods: `to_yaml_tree` and `from_yaml_tree` that construct the YAML content and convert the YAML content to Python class instances respectively.

A manifest file is used to associate tags and schema ID's so that if a schema has been defined, that the ASDF content can be validated against the schema (as well as providing extra information for the ASDF content in the info command). Normally the converters and manifest are registered with the ASDF library using standard functions, and this registration is normally (but is not required to be) triggered by use of Python entry points defined in the `setup.cfg` file so that this extension is automatically recognized when the extension package is installed.

One can of course write their own custom code to convert the contents of ASDF files however they want. The advantage of the tag/converter system is that the objects can be anywhere in the tree structure and be properly saved and recovered without having any implied knowledge of what attribute or location the object is at. Furthermore, it brings with it the ability to validate the contents by use of schema files.

Jupyter tutorials that show how to use converters can be found at:

- https://github.com/asdf-format/tutorials/blob/master/Your_first_ASDF_converter.ipynb
- https://github.com/asdf-format/tutorials/blob/master/Your_second_ASDF_converter.ipynb

ASDF Roadmap for STScI Work

The planned enhancements to ASDF are understandably focussed on the needs of STScI missions. Nevertheless, we are particularly interested in areas that have wider benefit to the general scientific and engineering community, and such considerations increase the priority of items necessary to STScI. Furthermore, we are eager to aid others working on ASDF by providing advice, reviews, and

```
├─meta (dict)
│ ├─aperture (Aperture) # Aperture information
│ │ ├─name (str): WFI_CEN # PRD science aperture used
│ │ └─position_angle (int): 120 # [deg] Position angle of aperture used
│ ├─cal_step (CalStep) # Calibration Status
│ │ ├─assign_wcs (str): COMPLETE # Assign World Coordinate System
│ │ ├─flat_field (str): COMPLETE # Flat Field Step
│ │ ├─dark (str): COMPLETE # Dark Subtraction
│ │ ├─dq_init (str): COMPLETE # Data Quality Mask Step
│ │ ├─jump (str): COMPLETE # Jump Detection Step
│ │ ├─linearity (str): COMPLETE # Linearity Correction
│ │ ├─photom (str): COMPLETE # Photometry Step
│ │ ├─ramp_fit (str): COMPLETE # Ramp Fitting
│ │ └─saturation (str): COMPLETE # Saturation Checking
│ ├─calibration_software_version (str): 0.4.3.dev89+gca5771d
│ ├─coordinates (Coordinates) # Information about the coordinates in the file
│ │ └─reference_frame (str): ICRS # Name of the coordinate reference frame
│ ├─crds_context_used (str): roman_0031.pmap
│ ├─crds_software_version (str): 11.5.0
│ ├─ephemeris (Ephemeris) # Ephemeris data information
│ │ ├─earth_angle (float): 3.3161255787892263 # [radians] Earth Angle
│ │ ├─moon_angle (float): 3.3196162372932148 # [radians] Moon Angle
│ │ ├─sun_angle (float): 3.316474644639625 # [radians] Sun Angle
│ │ ├─type (str): PREDICTED # Type of ephemeris
│ │ ├─time (float): 59215.0 # UTC time of position and velocity vectors in ephemeris (MJD)
│ │ ├─ephemeris_reference_frame (str): EME2000 # Ephemeris reference frame
│ │ ├─spatial_x (int): 100 # [km] X spatial coordinate of Roman
│ │ ├─spatial_y (int): 20 # [km] Y spatial coordinate of Roman
│ │ ├─spatial_z (int): 35 # [km] Z spatial coordinate of Roman
│ │ ├─velocity_x (int): 10 # [km/s] X component of Roman velocity
│ │ ├─velocity_y (int): 2 # [km/s] Y component of Roman velocity
│ │ └─velocity_z (float): 3.5 # [km/s] Z component of Roman velocity
│ ├─exposure (Exposure) # Exposure information

│ │ ├─id (int): 1 # Exposure id number within visit
│ │ ├─type (str): WFI_IMAGE
│ │ ├─start_time (Time) # UTC exposure start time
│ │ ├─mid_time (Time) # UTC exposure mid time
│ │ ├─end_time (Time) # UTC exposure end time
│ │ ├─start_time_mjd (float): 59215.0 # [d] exposure start time in MJD
│ │ ├─mid_time_mjd (float): 59215.000862037035 # [d] exposure mid time in MJD
│ │ ├─end_time_mjd (float): 59215.00172407407 # [d] exposure end time in MJD
```

Fig. 2: *This shows part of the output of the info command that shows the structure of a Roman Space Telescope test file (provided by the Roman Telescopes Branch at STScI). Displayed is the relative depth of the item, its type, value, and a title extracted from the associated schema to be used as explanatory information.*

possibly collaborative coding effort. STScI is committed to the long-term support of ADSF.

The following is a list of planned work, in order of decreasing priority.

Chunking Support

Since the Roman mission is expected to deal with large data sets and mosaicked images, support for chunking is considered essential. We expect to layer the support in our Python library on `zarr` [https://zarr.dev/], with two different representations, one where all data is contained within the ADSF file in separate blocks, and one where the blocks are saved in individual files. Both representations have important advantages and use cases.

Improvements to binary block management

These enhancements are needed to enable better chunking support and other capabilities.

Redefining versioning semantics

Previously the meaning of different levels of versioning were unclear. The normal inclination is to treat schema version using the typical semantic versioning system defined for software. But schemas are not software and we are inclined to use the proposed system for schemas [url: https://snowplowanalytics.com/blog/2014/05/13/introducing-schemaver-for-semantic-versioning-of-schemas/] To summarize: in this case the three levels of versioning correspond to:

Model.Revision.Addition where a schema change:

- [Model] prevents working with historical data
- [Revision] may prevent working with historical data
- [Addition] is compatible with all historical data

Integration into astronomy display tools

It is essential that astronomers be able to visualize the data contained within ASDF files conveniently using the commonly available tool, such as SAOImage DS9 [Joy03] and Ginga [Jes13].

Cloud optimized storage

Much of the future data processing operations for STScI are expected to be performed on the cloud, so having ASDF efficiently support such uses is important. An important element of this is making the format work efficiently with object storage services such as AWS S3 and Google Cloud Storage.

IDL support

While Python is rapidly surpassing the use of IDL in astronomy, there is still much IDL code being used, and many of those still using IDL are in more senior and thus influential positions (they aren't quite dead yet). So making ASDF data at least readable to IDL is a useful goal.

Support Rice compression

Rice compression [Pen09], [Pen10] has proven a useful lossy compression algorithm for astronomical imaging data. Supporting it will be useful to astronomers, particularly for downloading large imaging data sets.

Pandas Dataframe support

Pandas [McK10] has proven to be a useful tool to many astronomers, as well as many in the sciences and engineering, so support will enhance the uptake of ASDF.

Compact, easy-to-read schema summaries

Most scientists and even scientific software developers tend to find JSON Schema files tedious to interpret. A more compact, and intuitive rendering of the contents would be very useful.

Independent implementation

Having ASDF accepted as a standard data format requires a library that is divorced from a Python API. Initially this can be done most easily by layering it on the Python library, but ultimately there should be an independent implementation which includes support for C/C++ wrappers. This is by far the item that will require the most effort, and would benefit from outside involvement.

Provide interfaces to other popular packages

This is a catch all for identifying where there would be significant advantages to providing the ability to save and recover information in the ASDF format as an interchange option.

Sources of Information

- ASDF Standard: https://asdf-standard.readthedocs.io/en/latest/
- Python ASDF package documentation: https://asdf.readthedocs.io/en/stable/
- Repository: https://github.com//asdf-format/asdf
- Tutorials: https://github.com/asdf-format/tutorials

References

[Gre15] P. Greenfield, M. Droettboom, E. Bray. *ASDF: A new data format for astronomy*, Astronomy and Computing, 12:240-251, September 2015. https://doi.org/10.1016/j.ascom.2015.06.004

[FIT16] FITS Working Group. *Definition of the Flexible Image Transport System*, International Astronomical Union, http://fits.gsfc.nasa.gov/fits_standard.html, July 2016.

[Jes13] E. Jeschke. *Ginga: an open-source astronomical image viewer and toolkit*, Proc. of the 12th Python in Science Conference., p58-64,January 2013. https://doi.org/10.25080/Majora-8b375195-00a

[McK10] W. McKinney. *Data structures for statistical computing in python*, Proceedigns of the 9th Python in Science Conference, p56-61, 2010. https://doi.org/10.25080/Majora-92bf1922-00a

[Pen09] W. Pence, R. Seaman, R. L. White, *Lossless Astronomical Image Compression and the Effects of Noise*, Publications of the Astronomical Society of the Pacific, 121:414-427, April 2009. https://doi.org/10.48550/arXiv.0903.2140

[Pen10] W. Pence, R. L. White, R. Seaman. *Optimal Compression of Floating-Point Astronomical Images Without Significant Loss of Information*, Publications of the Astronomical Society of the Pacific, 122:1065-1076, September 2010. https://doi.org/10.1086/656249

[Joy03] W. A. Joye, E. Mandel. *New Features of SAOImage DS9*, Astronomical Data Analysis Software and Systems XII ASP Conference Series, 295:489, 2003.

Semi-Supervised Semantic Annotator (S3A): Toward Efficient Semantic Labeling

Nathan Jessurun[‡*], Daniel E. Capecci[‡], Olivia P. Dizon-Paradis[‡], Damon L. Woodard[‡], Navid Asadizanjani[‡]

Abstract—Most semantic image annotation platforms suffer severe bottlenecks when handling large images, complex regions of interest, or numerous distinct foreground regions in a single image. We have developed the Semi-Supervised Semantic Annotator (S3A) to address each of these issues and facilitate rapid collection of ground truth pixel-level labeled data. Such a feat is accomplished through a robust and easy-to-extend integration of arbitrary python image processing functions into the semantic labeling process. Importantly, the framework devised for this application allows easy visualization and machine learning prediction of arbitrary formats and amounts of per-component metadata. To our knowledge, the ease and flexibility offered are unique to S3A among all open-source alternatives.

Index Terms—Semantic annotation, Image labeling, Semi-supervised, Region of interest

Introduction

Labeled image data is essential for training, tuning, and evaluating the performance of many machine learning applications. Such labels are typically defined with simple polygons, ellipses, and bounding boxes (i.e., "this rectangle contains a cat"). However, this approach can misrepresent more complex shapes with holes or multiple regions as shown later in Figure 9. When high accuracy is required, labels must be specified at or close to the pixel-level - a process known as semantic labeling or semantic segmentation. A detailed description of this process is given in [CZF+18]. Examples can readily be found in several popular datasets such as COCO, depicted in Figure 1.

Semantic segmentation is important in numerous domains including printed circuit board assembly (PCBA) inspection (discussed later in the case study) [PJTA20], [AML+19], quality control during manufacturing [FRLL18], [AVK+01], [AAV+02], manuscript restoration / digitization [GNP+04], [KBO16], [JB92], [TFJ89], [FNK92], and effective patient diagnosis [SKM+10], [RLO+17], [YPH+06], [IGSM14]. In all these cases, imprecise annotations severely limit the development of automated solutions and can decrease the accuracy of standard trained segmentation models.

Quality semantic segmentation is difficult due to a reliance on large, high-quality datasets, which are often created by manually labeling each image. Manual annotation is error-prone, costly, and greatly hinders scalability. As such, several tools have been proposed to alleviate the burden of collecting these ground-truth labels [itL18]. Unfortunately, existing tools are heavily biased toward lower-resolution images with few regions of interest (ROI), similar to Figure 1. While this may not be an issue for some datasets, such assumptions are *crippling* for high-fidelity images with hundreds of annotated ROIs [LSA+10], [WYZZ09].

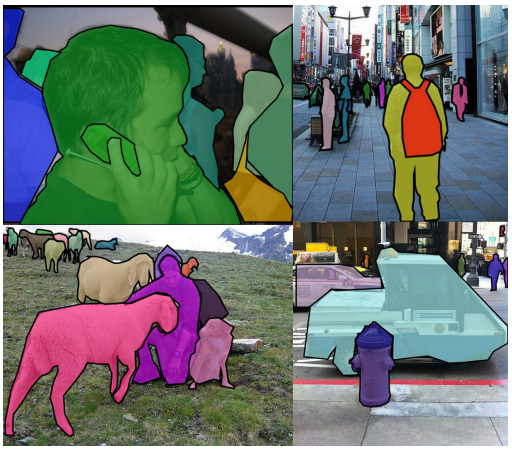


Fig. 1. Common use cases for semantic segmentation involve relatively few foreground objects, low-resolution data, and limited complexity per object. Images retrieved from https://cocodataset.org/#explore.

With improving hardware capabilities and increasing need for high-resolution ground truth segmentation, there are a continually growing number of applications that *require* high-resolution imaging with the previously described characteristics [MKS18], [DS20]. In these cases, the existing annotation tooling greatly impacts productivity due to the previously referenced assumptions and lack of support [Spa20].

In response to these bottlenecks, *we present the Semi-Supervised Semantic Annotation (S3A) annotation and prototyping platform -- an application which eases the process of pixel-level labeling in large, complex scenes.*[1] Its graphical user interface is shown in Figure 2. The software includes live app-level property customization, real-time algorithm modification and feedback, region prediction assistance, constrained component table editing based on allowed data types, various data export formats, and a highly adaptable set of plugin interfaces for domain-specific extensions to S3A. Beyond software improvements, these features play significant roles in bridging the gap between human annotation efforts and scalable, automated segmentation methods [BWS+10].

* *Corresponding author: njessurun@ufl.edu*
‡ *University of Florida*

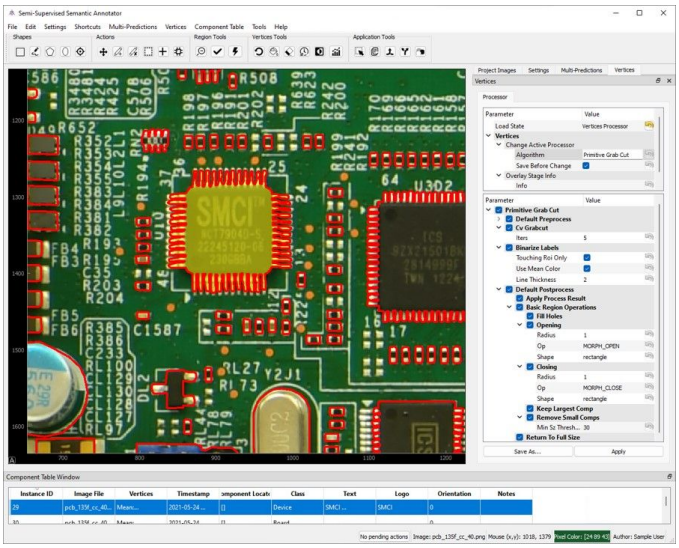


Fig. 2. S3A's interface. The main view consists of an image to annotate, a component table of prior annotations, and a toolbar which changes functionality depending on context.

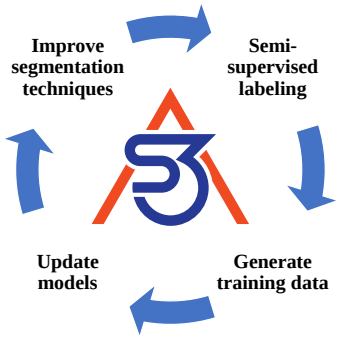


Fig. 3. S3A's can iteratively annotate, evaluate, and update its internals in real-time.

Application Overview

Design decisions throughout S3A's architecture have been driven by the following objectives:

- Metadata should have significance rather than be treated as an afterthought,
- High-resolution images should have minimal impact on the annotation workflow,
- ROI density and complexity should not limit annotation workflow, and
- Prototyping should not be hindered by application complexity.

These motives were selected upon noticing the general lack of solutions for related problems in previous literature and tooling. Moreover, applications that *do* address multiple aspects of complex region annotation often require an enterprise service and cannot be accessed under open-source policies.

While the first three points are highlighted in the case study, the subsections below outline pieces of S3A's architecture that prove useful for iterative algorithm prototyping and dataset generation as depicted in Figure 3. Note that beyond the facets illustrated here, S3A possesses multiple additional characteristics as outlined in its documentation (https://gitlab.com/s3a/s3a/-/wikis/docs/User's-Guide).

Processing Framework

At the root of S3A's functionality and configurability lies its adaptive processing framework. Functions exposed within S3A are thinly wrapped using a `Process` structure responsible for parsing signature information to provide documentation, parameter information, and more to the UI. Hence, all graphical depictions are abstracted beyond the concern of the user while remaining trivial to specify (but can be modified or customized if desired). As a result, incorporating additional/customized application functionality can require as little as one line of code. Processes interface with PyQtGraph parameters to gain access to data-customized widget types and more (https://github.com/pyqtgraph/pyqtgraph).

These processes can also be arbitrarily nested and chained, which is critical for developing hierarchical image processing models, an example of which is shown in Figure 4. This framework is used for all image and region processing within S3A. Note that for image processes, each portion of the hierarchy yields intermediate outputs to determine which stage of the process flow is responsible for various changes. This, in turn, reduces the effort required to determine which parameters must be adjusted to achieve optimal performance.

Plugins for User Extensions

The previous section briefly described how custom user functions are easily wrapped within a process, exposing its parameters within S3A in a GUI format. A rich plugin interface is built on top of this capability in which custom functions, table field predictors, default action hooks, and more can be directly integrated into S3A. In all cases, only a few lines of code are required to achieve most integrations between user code and plugin interface specifications. The core plugin infrastructure consists of a function/property registration mechanism and an interaction window that shows them in the UI. As such, arbitrary user functions can be "registered" in one line of code to a plugin, where it will be effectively exposed to the user within S3A. A trivial example is depicted in Figure 5, but more complex behavior such as OCR integration is possible with similar ease (see this snippet for an implementation leveraging `easyocr`).

Plugin features are heavily oriented toward easing the process of automation both for general annotation needs and niche datasets. In either case, incorporating existing library functions is converted into a trivial task directly resulting in lower annotation time and higher labeling accuracy.

Adaptable I/O

An extendable I/O framework allows annotations to be used in a myriad of ways. Out-of-the-box, S3A easily supports instance-level segmentation outputs, facilitating deep learning model training. As an example, Figure 6 illustrates how each instance in the image becomes its own pair of image and mask data. When several instances overlap, each is uniquely distinguishable depending on the characteristic of their label field. Particularly helpful for

1. A preliminary version was introduced in an earlier publication [JPRA20], but significant changes to the framework and tool capabilities have been employed since then.

Fig. 4. Outputs of each processing stage can be quickly viewed in context after an iteration of annotating. Upon inspecting the results, it is clear the failure point is a low k value during K-means clustering and segmentation. The woman's shirt is not sufficiently distinguishable from the background palette to denote a separate entity. The red dot is an indicator of where the operator clicked during annotation.

```python
from qtpy import QtWidgets
from s3a import (
    S3A,
    __main__,
    RandomToolsPlugin,
)

def hello_world(win: S3A):
    QtWidgets.QMessageBox.information(
        win, "Hello World", "Hello World!"
    )

RandomToolsPlugin.deferredRegisterFunc(
    hello_world
)

__main__.mainCli()
```

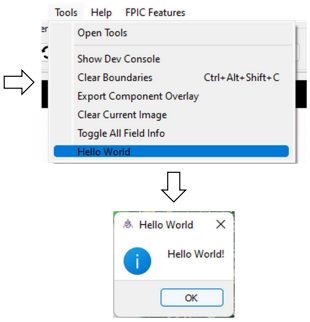


Fig. 5. Simple standalone functions can be easily exposed to the user through the random tools plugin. Note that if tunable parameters were included in the function signature, pressing "Open Tools" (the top menu option) allows them to be altered.

Fig. 6. Multiple export formats exist, among which is a utility that crops components out of the image, optionally padding with scene pixels and resizing to ensure all shapes are equal. Each sub-image and mask is saved accordingly, which is useful for training on multiple forms of machine learning models.

models with fixed input sizes, these exports can optionally be forced to have a uniform shape (e.g., 512x512 pixels) while maintaining their aspect ratio. This is accomplished by incorporating additional scene pixels around each object until the appropriate size is obtained. Models trained on these exports can be directly plugged back into S3A's processing framework, allowing them to generate new annotations or refine preliminary user efforts. The described I/O framework is also heavily modularized such that custom dataset specifications can easily be incorporated. In this manner, future versions of S3A will facilitate interoperability with popular formats such as COCO and Pascal VOC [LMB+14], [EGW+10].

Deep, Portable Customizability

Beyond the features previously outlined, S3A provides numerous avenues to configure shortcuts, color schemes, and algorithm workflows. Several examples of each can be seen in the user guide. Most customizable components prototyped within S3A can also be easily ported to external workflows after development. Hierarchical processes have states saved in YAML files describing all parameters, which can be reloaded to create user profiles. Alternatively, these same files can describe ideal parameter combinations for functions outside S3A in the event they are utilized in a different framework.

Case Study

Both the inspiration and developing efforts for S3A were initially driven by optical printed circuit board (PCB) assurance needs. In this domain, high-resolution images can contain thousands of complex objects in a scene, as seen in Figure 7. Moreover, numerous components are not representable by cardinal shapes such as rectangles, circles, etc. Hence, high-count polygonal regions dominated a significant portion of the annotated regions. The computational overhead from displaying large images and substantial numbers of complex regions either crashed most annotation platforms or prevented real-time interaction. In response, S3A was designed to fill the gap in open-source annotation platforms that addressed each issue while requiring minimal setup and allowing easy prototyping of arbitrary image processing tasks. The subsections below describe how the S3A labeling platform was utilized to collect a large database of PCB annotations along with their associated metadata[2].

Large Images with Many Annotations

In optical PCB assurance, one method of identifying component defects is to localize and characterize all objects in the image. Each

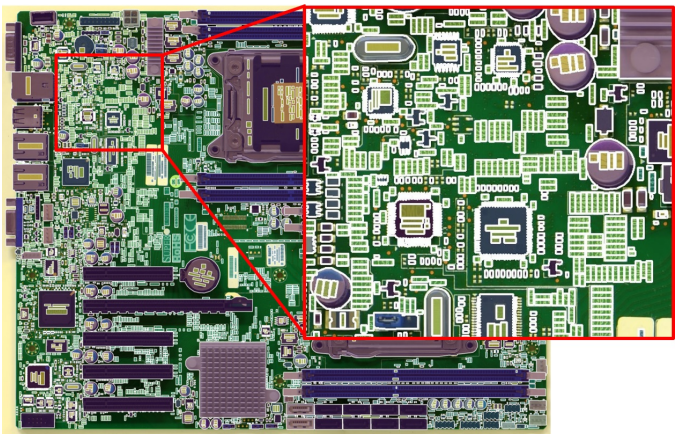


Fig. 7. Example PCB segmentation. In contrast to typical semgentation tasks, the scene contains over 4,000 objects with numerous complex shapes.

component can then be cross-referenced against genuine properties such as length/width, associated text, allowed orientations, etc. However, PCB surfaces can contain hundreds to thousands of components at several magnitudes of size, necessitating high-resolution images for in-line scanning. To handle this problem more generally, S3A separates the editing and viewing experiences. In other words, annotation time is orders of magnitude faster since only edits in one region at a time and on a small subset of the full image are considered during assisted segmentation. All other annotations are read-only until selected for alteration. For instance, Figure 8 depicts user inputs on a small ROI out of a much larger image. The resulting component shape is proposed within seconds and can either be accepted or modified further by the user. While PCB annotations initially inspired this approach, it is worth noting that the architectural approach applies to arbitrary domains of image segmentation.

Another key performance improvement comes from resizing the processed region to a user-defined maximum size. For instance, if an ROI is specified across a large portion of the image but the maximum processing size is 500x500 pixels, the processed area will be downsampled to a maximum dimension length of 500 before intensive algorithms are run. The final output will be upsampled back to the initial region size. In this manner, optionally sacrificing a small amount of output accuracy can drastically accelerate runtime performance for larger annotated objects.

Complex Vertices/Semantic Segmentation

Multiple types of PCB components possess complex shapes which might contain holes or noncontiguous regions. Hence, it is beneficial for software like S3A to represent these features inherently with a `ComplexXYVertices` object: that is, a collection of polygons which either describe foreground regions or holes. This is enabled by thinly wrapping `opencv`'s contour and hierarchy logic. Example components difficult to accomodate with single-polygon annotation formats are illustrated in Figure 9.

At the same time, S3A also supports high-count polygons with no performance losses. Since region edits are performed by image processing algorithms, there is no need for each vertex to be manually placed or altered by human input. Thus, such non-interactive shapes can simply be rendered as a filled path without a large number of event listeners present. This is the

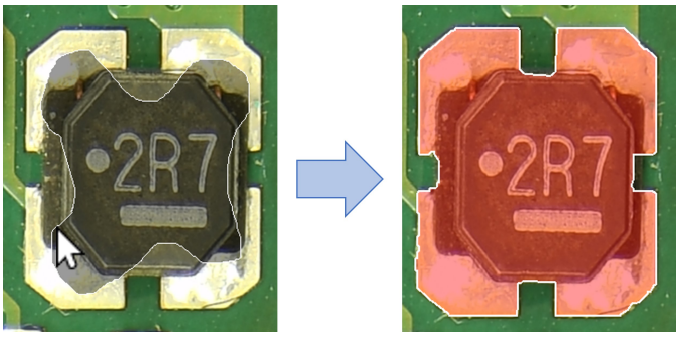


Fig. 8. Regardless of total image size and number of annotations, Python processing is be limited to the ROI or viewbox size for just the selected object based on user preferences. The depiction shows Grab Cut operating on a user-defined initial region within a much larger (8000x6000) image. The resulting region was available in 1.94 seconds on low-grade hardware.

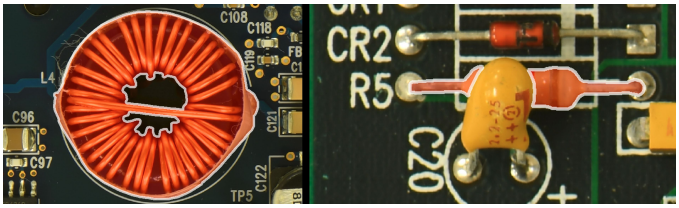


Fig. 9. Annotated objects in S3A can incorporate both holes and distinct regions through a multi-polygon container. Holes are represented as polygons drawn on top of existing foreground, and can be arbitrarily nested (i.e. island foreground is also possible).

key performance improvement when thousands of regions (each with thousands of points) are in the same field of view. When low polygon counts *are* required, S3A also supports RDP polygon simplification down to a user-specified epsilon parameter [Ram].

Complex Metadata

Most annotation software support robust implementation of image region, class, and various text tags ("metadata"). However, this paradigm makes collecting type-checked or input-sanitized metadata more difficult. This includes label categories such as object rotation, multiclass specifications, dropdown selections, and more. In contrast, S3A treats each metadata field the same way as object vertices, where they can be algorithm-assisted, directly input by the user, or part of a machine learning prediction framework. Note that simple properties such as text strings or numbers can be directly input in the table cells with minimal need for annotation assistance[3]. In conrast, custom fields can provide plugin specifications which allow more advanced user interaction. Finally, auto-populated fields like annotation timestamp or author can easily be constructed by providing a factory function instead of default value in the parameter specification.

This capability is particularly relevant in the field of optical PCB assurance. White markings on the PCB surface, known as silkscreen, indicate important aspects of nearby components. Thus, understanding the silkscreen's orientation, alphanumeric characters, associated component, logos present, and more provide several methods by which to characterize / identify features of their respective devices. Both default and customized input validators were applied to each field using parameter specifications, custom plugins, or simple factories as described above. A summary of the metadata collected for one component is shown in Figure 10.

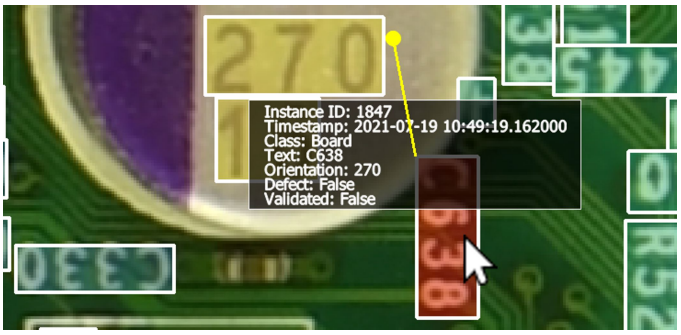


Fig. 10. Metadata can be collected, validated, and customized with ease. A mix of default properties (strings, numbers, booleans), factories (timestamp, author), and custom plugins (yellow circle representing associated device) are present.

Conclusion and Future Work

The Semi-Supervised Semantic Annotator (S3A) is proposed to address the difficult task of pixel-level annotations of image data. For high-resolution images with numerous complex regions of interest, existing labeling software faces performance bottlenecks attempting to extract ground-truth information. Moreover, there is a lack of capabilities to convert such a labeling workflow into an automated procedure with feedback at every step. Each of these challenges is overcome by various features within S3A specifically designed for such tasks. As a result, S3A provides not only tremendous time savings during ground truth annotation, but also allows an annotation pipeline to be directly converted into a prediction scheme. Furthermore, the rapid feedback accessible at every stage of annotation expedites prototyping of novel solutions to imaging domains in which few examples of prior work exist. Nonetheless, multiple avenues exist for improving S3A's capabilities in each of these areas. Several prominent future goals are highlighted in the following sections.

Dynamic Algorithm Builder

Presently, processing workflows can be specified in a sequential YAML file which describes each algorithm and their respective parameters. However, this is not easy to adapt within S3A, especially by inexperienced annotators. Future iterations of S3A will incoroprate graphical flowcharts which make this process drastically more intuitive and provide faster feedback. Frameworks like Orange [DCE+] perform this task well, and S3A would strongly benefit from adding the relevant capabilities.

Image Navigation Assistance

Several aspects of image navigation can be incorporated to simplify the handling of large images. For instance, a "minimap" tool would allow users to maintain a global image perspective while making local edits. Furthermore, this sense of scale aids intuition of how many regions of similar component density, color, etc. exist within the entire image.

Second, multiple strategies for annotating large images leverage a windowing approach, where they will divide the total image into several smaller pieces in a gridlike fashion. While this has its disadvantages, it is fast, easy to automate, and produces reasonable results depending on the initial image complexity [VGSG+19]. Hence, these methods would be significantly easier to incorporate into S3A if a generalized windowing framework was incorporated which allows users to specify all necessary parameters such as window overlap, size, sampling frequency, etc. A preliminary version of this is implemented for categorical-based model prediction, but a more robust feature set for interactive segmentation is strongly preferable.

Aggregation of Human Annotation Habits

Several times, it has been noted that manual segmentation of image data is not a feasible or scalable approach for remotely large datasets. However, there are multiple cases in which human intuition can greatly outperform even complex neural networks, depending on the specific segmentation challenge [RLFF15]. For this reason, it would be ideal to capture data points possessing information about the human decision-making process and apply them to images at scale. This may include taking into account human labeling time per class, hesitation between clicks, relationship between shape boundary complexity and instance quantity, and more. By aggregating such statistics, a pattern may arise which can be leveraged as an additional automated annotation technique.

2. For those curious, the dataset and associated paper are accessible at https://www.trust-hub.org/#/data/pcb-images.

3. For a list of input validators and supported primitive types, refer to PyQtGraph's Parameter documentation.

References

[AAV+02] C Anagnostopoulos, I Anagnostopoulos, D Vergados, G Kouzas, E Kayafas, V Loumos, and G Stassinopoulos. High performance computing algorithms for textile quality control. *Mathematics and Computers in Simulation*, 60(3):389–400, September 2002. doi:10.1016/S0378-4754(02)00031-9.

[AML+19] Mukhil Azhagan, Dhwani Mehta, Hangwei Lu, Sudarshan Agrawal, Mark Tehranipoor, Damon L Woodard, Navid Asadizanjani, and Praveen Chawla. A review on automatic bill of material generation and visual inspection on PCBs. In *ISTFA 2019: Proceedings of the 45th International Symposium for Testing and Failure Analysis*, page 256. ASM International, 2019.

[AVK+01] C. Anagnostopoulos, D. Vergados, E. Kayafas, V. Loumos, and G. Stassinopoulos. A computer vision approach for textile quality control. *The Journal of Visualization and Computer Animation*, 12(1):31–44, 2001. doi:10.1002/vis.245.

[BWS+10] Steve Branson, Catherine Wah, Florian Schroff, Boris Babenko, Peter Welinder, Pietro Perona, and Serge Belongie. Visual recognition with humans in the loop. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 438–451, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[CZF+18] Qimin Cheng, Qian Zhang, Peng Fu, Conghuan Tu, and Sen Li. A survey and analysis on automatic image annotation. *Pattern Recognition*, 79:242–259, 2018. doi:10.1016/j.patcog.2018.02.017.

[DCE+] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, and Anže Starič. Orange: Data mining toolbox in Python. 14(1):2349–2353.

[DS20] Polina Demochkina and Andrey V. Savchenko. Improving the accuracy of one-shot detectors for small objects in x-ray images. In *2020 International Russian Automation Conference (RusAutoCon)*, page 610–614. IEEE, September 2020. URL: https://ieeexplore.ieee.org/document/9208097/, doi:10.1109/RusAutoCon49822.2020.9208097.

[EGW+10] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, jun 2010. URL: https://doi.org/10.1007/s11263-009-0275-4, doi:10.1007/s11263-009-0275-4.

[FNK92] H. Fujisawa, Y. Nakano, and K. Kurino. Segmentation methods for character recognition: From segmentation to document structure analysis. *Proceedings of the IEEE*, 80(7):1079–1092, July 1992. doi:10.1109/5.156471.

[FRLL18] Max K. Ferguson, Ak Ronay, Yung-Tsun Tina Lee, and Kincho. H. Law. Detection and segmentation of manufacturing defects with convolutional neural networks and transfer learning. *Smart and sustainable manufacturing systems*, 2, 2018. doi:10.1520/SSMS20180033.

[GNP+04] Basilios Gatos, Kostas Ntzios, Ioannis Pratikakis, Sergios Petridis, T. Konidaris, and Stavros J. Perantonis. A segmentation-free recognition technique to assist old greek handwritten manuscript OCR. In Simone Marinai and Andreas R. Dengel, editors, *Document Analysis Systems VI*, Lecture Notes in Computer Science, pages 63–74, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-28640-0_7.

[IGSM14] D. K. Iakovidis, T. Goudas, C. Smailis, and I. Maglogiannis. Ratsnake: A versatile image annotation tool with application to computer-aided diagnosis, 2014. doi:10.1155/2014/286856.

[itL18] Humans in the Loop. The best image annotation platforms for computer vision (+ an honest review of each), October 2018. URL: https://hackernoon.com/the-best-image-annotation-platforms-for-computer-vision-an-honest-review-of-each-dac7f565fea.

[JB92] Anil K. Jain and Sushil Bhattacharjee. Text segmentation using gabor filters for automatic document processing. *Machine Vision and Applications*, 5(3):169–184, June 1992. doi:10.1007/BF02626996.

[JPRA20] Nathan Jessurun, Olivia Paradis, Alexandra Roberts, and Navid Asadizanjani. Component Detection and Evaluation Framework (CDEF): A Semantic Annotation Tool. *Microscopy and Microanalysis*, 26(S2):1470–1474, August 2020. doi:10.1017/S1431927620018243.

[KBO16] Made Windu Antara Kesiman, Jean-Christophe Burie, and Jean-Marc Ogier. A new scheme for text line and character segmentation from gray scale images of palm leaf manuscript. In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 325–330, October 2016. doi:10.1109/ICFHR.2016.0068.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[LSA+10] L'ubor Ladický, Paul Sturgess, Karteek Alahari, Chris Russell, and Philip H. S. Torr. What, where and how many? combining object detectors and crfs. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 424–437, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[MKS18] S. Mohajerani, T. A. Krammer, and P. Saeedi. A cloud detection algorithm for remote sensing images using fully convolutional neural networks. In *2018 IEEE 20th International Workshop on Multimedia Signal Processing (MMSP)*, page 1–5, August 2018. doi:10.1109/MMSP.2018.8547095.

[PJTA20] Olivia P Paradis, Nathan T Jessurun, Mark Tehranipoor, and Navid Asadizanjani. Color normalization for robust automatic bill of materials generation and visual inspection of pcbs. In *ISTFA 2020: Papers Accepted for the Planned 46th International Symposium for Testing and Failure Analysis*, International Symposium for Testing and Failure Analysis, pages 172–179, 2020. URL: https://doi.org/10.31399/asm.cp.istfa2020p0172https://dl.asminternational.org/istfa/proceedings-pdf/ISTFA2020/83348/172/425605/istfa2020p0172.pdf, doi:10.31399/asm.cp.istfa2020p0172.

[Ram] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. 1(3):244–256. URL: https://www.sciencedirect.com/science/article/pii/S0146664X72800170, doi:10.1016/S0146-664X(72)80017-0.

[RLFF15] Olga Russakovsky, Li-Jia Li, and Li Fei-Fei. Best of both worlds: Human-machine collaboration for object annotation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 2121–2131. IEEE, June 2015. URL: http://ieeexplore.ieee.org/document/7298824/, doi:10.1109/CVPR.2015.7298824.

[RLO+17] Martin Rajchl, Matthew C. H. Lee, Ozan Oktay, Konstantinos Kamnitsas, Jonathan Passerat-Palmbach, Wenjia Bai, Mellisa Damodaram, Mary A. Rutherford, Joseph V. Hajnal, Bernhard Kainz, and Daniel Rueckert. DeepCut: Object segmentation from bounding box annotations using convolutional neural networks. *IEEE Transactions on Medical Imaging*, 36(2):674–683, February 2017. doi:10.1109/TMI.2016.2621185.

[SKM+10] Sascha Seifert, Michael Kelm, Manuel Moeller, Saikat Mukherjee, Alexander Cavallaro, Martin Huber, and Dorin Comaniciu. Semantic annotation of medical images. In Brent J. Liu and William W. Boonn, editors, *Medical Imaging 2010: Advanced PACS-based Imaging Informatics and Therapeutic Applications*, volume 7628, pages 43 – 50. International Society for Optics and Photonics, SPIE, 2010. URL: https://doi.org/10.1117/12.844207, doi:10.1117/12.844207.

[Spa20] SpaceNet. Multi-Temporal Urban Development Challenge. https://spacenet.ai/sn7-challenge/, June 2020.

[TFJ89] T. Taxt, P.J. Flynn, and A.K. Jain. Segmentation of document images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(12):1322–1329, December 1989. doi:10.1109/34.41371.

[VGSG+19] Juan P. Vigueras-Guillén, Busra Sari, Stanley F. Goes, Hans G. Lemij, Jeroen van Rooij, Koenraad A. Vermeer, and Lucas J. van Vliet. Fully convolutional architecture vs sliding-window cnn for corneal endothelium cell segmentation. *BMC Biomedical Engineering*, 1(1):4, January 2019. doi:10.1186/s42490-019-0003-2.

[WYZZ09] C. Wang, Shuicheng Yan, Lei Zhang, and H. Zhang. Multi-label sparse coding for automatic image annotation. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, page 1643–1650, June 2009. doi:10.1109/CVPR.2009.5206866.

[YPH+06] Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. *NeuroImage*, 31(3):1116–1128, July 2006. doi:10.1016/j.neuroimage.2006.01.015.

Galyleo: A General-Purpose Extensible Visualization Solution

Rick McGeer‡*, Andreas Bergen‡, Mahdiyar Biazi‡, Matt Hemmings‡, Robin Schreiber‡

Abstract—Galyleo is an open-source, extensible dashboarding solution integrated with JupyterLab [jup]. Galyleo is a standalone web application integrated as an iframe [LS10] into a JupyterLab tab. Users generate data for the dashboard inside a Jupyter Notebook [KRKP+16], which transmits the data through message passing [mdn] to the dashboard; users use drag-and-drop operations to add widgets to filter, and charts to display the data, shapes, text, and images. The dashboard is saved as a JSON [Cro06] file in the user's filesystem in the same directory as the Notebook.

Index Terms—JupyterLab, JupyterLab extension, Data visualization

Introduction

Current dashboarding solutions [hol22a] [hol22b] [plo] [pan22] for Jupyter either involve external, heavyweight tools, ingrained HTML/CSS coding, complex publication, or limited control over layout, and have restricted widget sets and visualization libraries. Graphics objects require a great deal of configuration: size, position, colors, fonts must be specified for each object. Thus library solutions involve a significant amount of fairly simple code. Conversely, visualization involves analytics, an inherently complex set of operations. Visualization tools such as Tableau [DGHP13] or Looker [loo] combine visualization and analytics in a single application presented through a point-and-click interface. Point-and-click interfaces are limited in the number and complexity of operations supported. The complexity of an operation isn't reduced by having a simple point-and-click interface; instead, the user is confronted with the challenge of trying to do something complicated by pointing. The result is that tools encapsulate complex operations in a few buttons, and that leads to a limited number of operations with reduced options and/or tools with steep learning curves.

In contrast, Jupyter is simply a superior analytics environment in every respect over a standalone visualization tool: its various kernels and their libraries provide a much broader range of analytics capabilities; its programming interface is a much cleaner and simpler way to perform complex operations; hardware resources can scale far more easily than they can for a visualization tool; and connectors to data sources are both plentiful and extensible.

Both standalone visualization tools and Jupyter libraries have a limited set of visualizations. Jupyter is a *server-side platform*. Jupyter's web interface is primarily to offer textboxes for code entry. Entered code is sent to the server for evaluation and text/HTML results returned. Visualization in a Jupyter Notebook is either given by images rendered server-side and returned as inline image tags, or by JavaScript/HTML5 libraries which have a corresponding server-side Python library. The Python library generates HTML5/JavaScript code for rendering.

The limiting factor is that the visualization library must be integrated with the Python backend by a developer, and only a subset of the rich array of visualization, charting, and mapping libraries available on the HTML5/JavaScript platform is integrated. The HTML5/JavaScript platform is as rich a client-side visualization platform as Python is a server-side platform.

Galyleo set out to offer the best of both worlds: Python, R, and Julia as a scalable analytics platform coupled with an extensible JavaScript/HTML5 visualization and interaction platform. It offers a *no-code* client-side environment, for several reasons.

1) The Jupyter analytics community is comfortable with server-side analytics environments (the 100+ kernels available in Jupyter, including Python, R and Julia) but less so with the JavaScript visualization platform.
2) Configuration of graphical objects takes a lot of low-value configuration code; conversely, it is relatively easy to do by hand.

These insights lead to a mixed interface, combining a drag-and-drop interface for the design and configuration of visual objects, and a coding, server-side interface for analytics programs.

Extension of the widget set was an important consideration. A widget is a client-side object with a physical component. Galyleo is designed to be extensible both by adding new visualization libraries and components and by adding new widgets.

Publication of interactive dashboards has been a further challenge. A design goal of Galyleo was to offer a simple scheme, where a dashboard could be published to the web with a single click.

These then, are the goals of Galyleo:

1) Simple, drag-and-drop design of interactive dashboards in a visual editor. The visual design of a Galyleo dashboard should be no more complex than design of a PowerPoint or Google slide;
2) Radically simplify the dashboard-design interface by coupling it to a powerful, Jupyter back end to do the analytics work, separating visualization and analytics concerns;

* Corresponding author: rick.mcgeer@engageLively.com
‡ engageLively

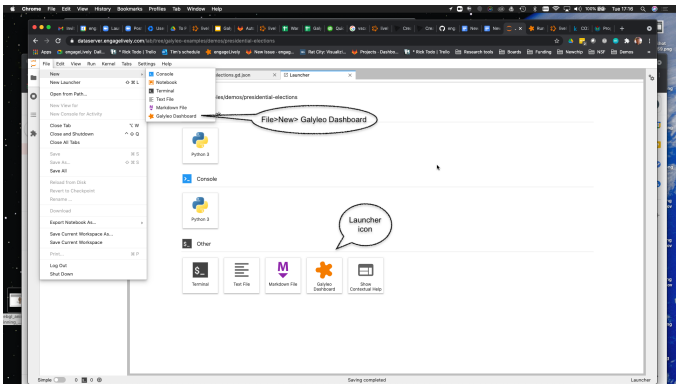


Fig. 1: *Figure 1. A New Galyleo Dashboard*

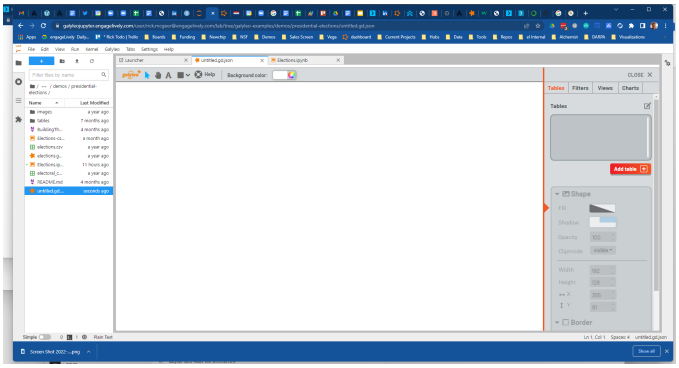


Fig. 2: *Figure 2. The Galyleo Dashboard Studio*

3) Maximimize extensibility for visualization and widgets on the client side and analytics libraries, data sources and hardware resources on the server side;
4) Easy, simple publication;

Using Galyleo

The general usage model of Galyleo is that a Notebook is being edited and executed in one tab of JupyterLab, and a corresponding dashboard file is being edited and executed in another; as the Notebook executes, it uses the Galyleo Client library to send data to the dashboard file. To JupyterLab, the Galyleo Dashboard Studio is just another editor; it reads and writes *.gd.json* files in the current directory.

The Dashboard Studio

A new Galyleo Dashboard can be launched from the JupyterLab launcher or from the File>New menu, as shown in Figure 1.

An existing dashboard is saved as a *.gd.json* file, and is denoted with the Galyleo star logo. It can be opened in the usual way, with a double-click.

Once a file is opened, or a new file created, a new Galyleo tab opens onto it. It resembles a simplified form of a Tableau, Looker, or PowerBI editor. The collapsible right-hand sidebar offers the ability to view Tables, and view, edit, or create Views, Filters, and Charts. The bottom half of the right sidebar gives controls for styling of text and shapes.

The top bar handles the introduction of decorative and styling elements to the dashboard: labels and text, simple shapes such as ellipses, rectangles, polygons, lines, and images. All images are referenced by URL.

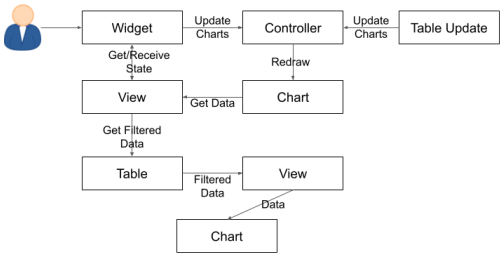


Fig. 3: *Figure 3. Dataflow in Galyleo*

As the user creates and manipulates the visual elements, the editor continuously saves the table as a JSON file, which can also be edited with Jupyter's built-in text editor.

Workflow

The goal of Galyleo is simplicity and transparency. Data preparation is handled in Jupyter, and the basic abstract item, the *GalyleoTable* is generally created and manipulated there, using an open-source Python library. When a table is ready, the *GalyleoClient* library is invoked to send it to the dashboard, where it appears in the table tab of the sidebar. The dashboard author then creates visual elements such as sliders, lists, dropdowns *etc.*, which select rows of the table, and uses these filtered lists as inputs to charts. The general idea is that the author should be able to seamlessly move between manipulating and creating data tables in the Notebook, and filtering and visualizing them in the dashboard.

Data Flow and Conceptual Picture

The Galyleo Data Model and Architecture is discussed in detail below. The central idea is to have a few, orthogonal, easily-grasped concepts which make data manipulation easy and intuitive. The basic concepts are as follows:

1) *Table*: A Table is a list of records, equivalent to a Pandas DataFrame [pdt20] [WM10] or a SQL Table. In general, in Galyleo, a Table is expected to be produced by an external source, generally a Jupyter Notebook
2) *Filter*: A Filter is a logical function which applies to a single column of a Table Table, and selects rows from the Table. Each Filter corresponds to a widget; widgets set the values Filter use to select Table rows
3) *View* A View is a subset of a Table selected by one or more Filters. To create a view, the user chooses a Table, and then chooses one or more Tilters to apply to the Table to select the rows for the View. The user can also statically select a subset of the columns to include in the View.
4) *Chart* A Chart is a generic term for an object that displays data graphically. Its input is a View or a Table. Each Chart has a single data source.

The data flow is straightforward. A Table is updated from an external source, or the user manipulates a widget. When this happens, the affected item signals the dashboard controller that it has been updated. The controller then signals all charts to redraw themselves. Each Chart will then request updated data from its

source Table or View. A View then requests its configured filters for their current logic functions, and passes these to the source Table with a request to apply the filters and return the rows which are selected by *all* the filters (in the future, a more general Boolean will be applied; the UI elements to construct this function are under design). The Table then returns the rows which pass the filters; the View selects the static subset of columns it supports, and passes this to its Charts, which then redraw themselves.

Each item in this flow conceptually has a single data source, but multiple data targets. There can be multiple Views over a Table, but each View has a single Table as a source. There can be multiple charts fed by a View, but each Chart has a single Table or View as a source.

It's important to note that there are no special cases. There is no distinction, as there is in most visualization systems, between a "Dimension" or a "Measure"; there are simply columns of data, which can be either a value or category axis for any Chart. From this simplicity, significant generality is achieved. For example, a filter selects values from any column, whether that column is providing value or category. Applying a range filter to a category column gives natural telescoping and zooming on the x-axis of a chart, without change to the architecture.

Drilldowns

An important operation for any interactive dashboard is drill-downs: expanding detail for a datapoint on a chart. The user should be able to click on a chart and see a detailed view of the data underlying the datapoint. This was naturally implemented in our system by associating a filter with every chart: *every chart in Galyleo is also a Select Filter, and it can be used as a Filter in a view, just as any other widget can be*.

Publishing The Dashboard

Once the dashboard is complete, it can be published to the web simply by moving the dashboard file to any place it get an URL (e.g. a github repo). It can then be viewed by visiting *https://galyleobeta.engagelively.com/public/galyleo/index.html?dashboard=<url of dashboard file>*. The attached figure shows a published Galyleo Dashboard, which displays Florence Nightingale's famous Crimean War dataset. Using the double sliders underneath the column charts telescope the x axes, effectively permitting zooming on a range; clicking on a column shows the detailed death statistics for that month in the pie chart above the column chart.

No-Code, Low-Code, and Appropriate-Code

Galyleo is an appropriate-code environment, meaning that it offers efficient creation to developers at every step. It offers What-You-See-Is-What-You-Get (WYSIWYG) design tools where appropriate, low-code where appropriate, and full code creation tools where appropriate.

No-code and low-code environments, where users construct applications through a visual interface, are popular for several reasons. The first is the assumption that coding is time-consuming and hard, which isn't always or necessarily true; the second is the assumption that coding is a skill known to only a small fraction of the population, which is becoming less true by the day. 40% of Berkeley undergraduates take Data 8, in which every assignment involves programming in a Jupyter Notebook. The third, particularly for graphics code, is that manual design

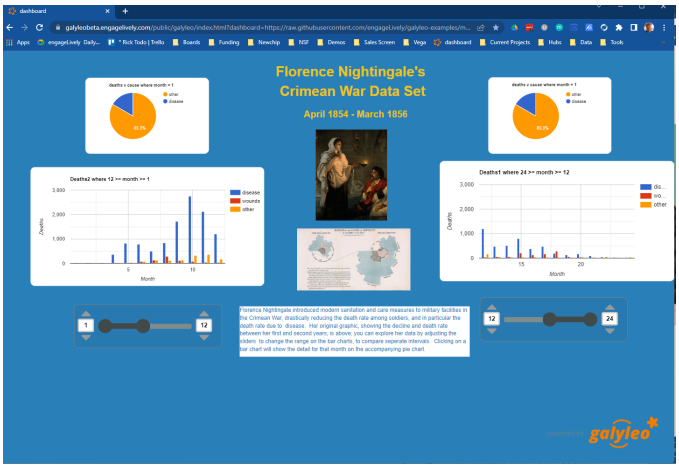


Fig. 4: *Figure 4. A Published Galyleo Dashboard*

and configuration gives instant feedback and tight control over appearance. For example, the authors of a LaTeX paper (including this one) can't control the placement of figures within the text. The fourth, which is correct, is that configuration code is more verbose, error-prone, and time-consuming than manual configuration.

What is less often appreciated is that when operations become sufficiently complex, coding is a much simpler interface than manual configuration. For example, building a pivot table in a spreadsheet using point-and-click operations have "always had a reputation for being complicated" [Dev]. It's three lines of code in Python, even without using the Pandas *pivot_table* method. Most analytics procedures are far more easily done in code.

As a result, Galyleo is an *appropriate-code* environment, which is an environment which combines a coding interface for complex, large-scale, or abstract operations and a point-and-click interface for simple, concrete, small-scale operations. Galyleo combines broadly powerful Jupyter-based code and low-code libraries for analytics paired with fast GUI-based design and configuration for graphical elements and layout.

Galyleo Data Model And Architecture

The Galyleo data Model and architecture closely model the dashboard architecture discussed in the previous section. They are based on the idea of a few simple, generalizable structures, which are largely independent of each other and communicate through simple interfaces.

The GalyleoTable

A GalyleoTable is the fundamental data structure in Galyleo. It is a logical, not a physical abstraction; it simply responds to the GalyleoTable API. A GalyleoTable is a pair *(columns, rows)*, where *columns* is a list of pairs *(name, type)*, where *type* is one of *{string, boolean, number, date}*, and *rows* is a list of lists of primitive values, where the length of each component list is the length of the list of columns and the type of the kth entry in each list is the type specified by the kth column.

Small, public tables may be contained in the dashboard file; these are called *explicit* tables. However, explicitly representing the table in the dashboard file has a number of disadvantages:

1) An explicit table is in the memory of the client viewing the dashboard; if it is too large, it may cause signifi-

cant performance problems on the dashboard author or viewer's device
2) Since the dashboard file is accessible on the web, any data within it is public
3) The data may be continuously updated from a source, and it's inconvenient to re-run the Notebook to update the data.

Therefore, the GalyleoTable can be of one of three types:

1) A data server that implements the Table REST API
2) A JavaScript object within the dashboard page itself
3) A JavaScript messenger in the page that implements a messaging version of the API

An explicit table is simply a special case of (2) -- in this case, the JavaScript object is simply a linear list of rows.

These are not exclusive. The JavaScript messenger case is designed to support the ability of a containing application within the browser to handle viewer authentication, shrinking the security vulnerability footprint and ensuring that the client application controls the data going to the dashboard. In general, aside from performing tasks like authentication, the messenger will call an external data server for the values themselves.

Whether in a Data Server, a containing application, or a JavaScript object, Tables support three operations:

1) Get all the values for a specific column
2) Get the max/min/increment for a specific numeric column
3) Get the rows which match a boolean function, passed in as a parameter to the operation

Of course, (3) is the operation that we have seen above, to populate a view and a chart. (1) and (2) populate widgets on the dashboard; (1) is designed for a select filter, which is a widget that lets a user pick a specific set of values for a column; (2) is an optimization for numeric filters, so that the entire list of values for the column need not be sent -- rather, only the start and end values, and the increment between them.

Each type of table specifies a source, additional information (in the case of a data server, for example, any header variables that must be specified in order to fetch the data), and, optionally, a polling interval. The latter is designed to handle live data; the dashboard will query the data source at each polling interval to see if the data has changed.

The choice of these three table instantiations (REST, JavaScript object, messenger) is that they provide the key foundational building block for future extensions; it's easy to add a SQL connection on top of a REST interface, or a Python simulator.

Filters

Tables must be filtered *in situ*. One of the key motivators behind remote tables is in keeping large amounts of data from hitting the browser. This is largely defeated if the entire table is sent to the dashboard and then filtered there. As a result, there is a Filter API together with the Table API whereever there are tables.

The data flow of the previous section remains unchanged; it is simply that the filter functions are transmitted to wherever the tables happen to be. The dataflow in the case of remote tables (whether messenger-based or REST-based) is shown here, with operations that are resident where the table is situated and operations resident on the dashboard clearly shown.

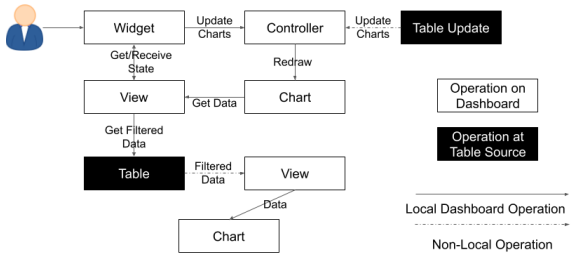


Fig. 5: *Figure 5. Galyleo Dataflow with Remote Tables*

Comments

Again, simplicity and orthogonality have shown tremendous benefits here. Though filters conceptually act as selectors on rows, they may perform a variety of roles in implementations. For example, a table produced by a simulator may be controlled by a parameter value given by a Filter function.

Extending Galyleo

Every element of the Galyleo system, whether it is a widget, Chart, Table Server, or Filter is defined exclusively through a small set of public APIs. This is done to permit easy extension, by either the Galyleo team, users, or third parties. A Chart is defined as an object which has a physical HTML representation, and it supports four JavaScript methods: redraw (draw the chart), set data (set the chart's data), set options (set the chart's options), and supports table (a boolean which returns true if and only if the chart can draw the passed-in data set). In addition, it exports out a defined JSON structure which indicates what options it supports and the types of their values; this is used by the Chart Editor to display a configurator for the chart.

Similarly, the underlying lively.next system supports user design of new filters. Again, a filter is simply an object with a physical presence, that the user can design in lively, and supports a specific API -- broadly, set the choices and hand back the Boolean function as a JSON object which will be used to filter the data.

lively.next

Any system can be used to extend Galyleo; at the end of the day, all that need be done is encapsulate a widget or chart in a snippet of HTML with a JavaScript interface that matches the Galyleo protocol. This is done most easily and quickly by using lively.next [SKH21]. lively.next is the latest in a line of Smalltalk- and Squeak-inspired [IKM+97] JavaScript/HTML integrated development environments that began with the Lively Kernel [IPU+08] [KIH+09] and continued through the Lively Web [LKI+12] [IFH+16] [TM17]. Galyleo is an application built in Lively, following the work done in [HIK+16].

Lively shares with Jupyter an emphasis on live programming [KRB18], orwhere a Read-Evaluate-Act Loop (REAL) programming style. It adds to that a combination of visual and text programming [ABF20], where physical objects are positioned and configured largely by hand as done with any drawing or design program (e.g., PowerPoint, Illustrator, DrawPad, Google Draw) and programmed with a built-in editor and workspace, similar in concept if not form to a Jupyter Notebook.

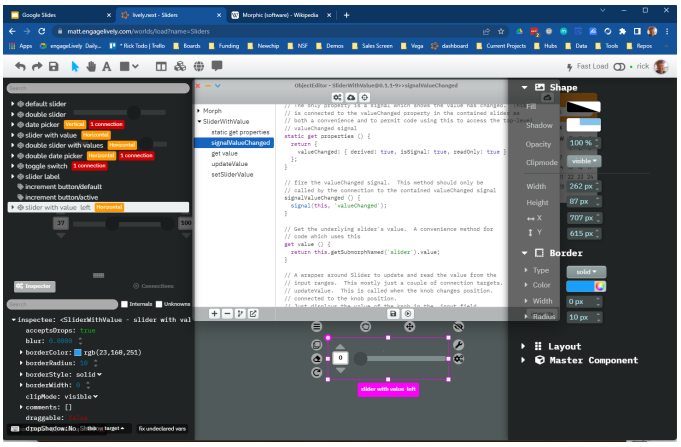


Fig. 6: *Figure 6. The lively.next environment*

Lively abstracts away HTML and CSS tags in graphical objects called "Morphs". Morphs [MS95] were invented as the user interface layer for Self [US87], and have been used as the foundation of the graphics system in Squeak and Scratch [MRR+10]. In this UI, every physical object is a Morph; these can be as simple as a simple polygon or text string to a full application. Morphs are combined via composition, similar to the way that objects are grouped in a presentation or drawing program. The composition is simply another Morph, which in turn can be composed with other Morphs. In this manner, complex Morphs can be built up from collections of simpler ones. For example, a slider is simply the composition of a circle (the knob) with a thin, long rectangle (the bar). Each Morph can be individually programmed as a JavaScript object, or can inherit base level behavior and extend it.

In lively.next, each morph turns into a snippet of HTML, CSS, and JavaScript code and the entire application turns into a web page. The programmer doesn't see the HTML and CSS code directly; these are auto-generated. Instead, the programmer writes JavaScript code for both logic and configuration (to the extent that the configuration isn't done by hand). The code is bundled with the object and integrated in the web page.

Morphs can be set as reusable components by a simple declaration. They can then be reused in any lively design.

Incorporating New Libraries

Libraries are typically incorporated into lively.next by attaching them to a convenient physical object, importing the library from a package manager such as npm, and then writing a small amount of code to expose the object's API. The simplest form of this is to assign the module to an instance variable so it has an addressable name, but typically a few convenience methods are written as well. In this way, a large number of libraries have been incorporated as reusable components in lively.next, including Google Maps, Google Charts [goo], Chartjs [cha], D3 [BOH11], Leaflet.js [lea], OpenLayers [ope], cytoscape:*ono* and many more.

Extending Galyleo's Charting and Visualization capabilities

A Galyleo Chart is anything that changes its display based on tabular data from a Galyleo Table or Galyleo View. It responds to a specific API, which includes two principal methods:

1) *drawChart*: redraw the chart using the current tabular data from the input or view
2) *acceptsDataset(<Table or View>)* returns a boolean depending on whether this chart can draw the data in this view. For example, a Table Chart can draw any tabular data; a Geo Chart typically requires that the first column be a place specifier.

In addition, it has a read-only property:

1) *optionSpec*: A JSON structure describing the options for the chart. This is a dictionary, which specifies the name of each option, and its type (color, number, string, boolean, or enum with values given). Each type corresponds to a specific UI widget that the chart editor uses.

And two read write properties:

1) *options*: The current options, as a JSON dictionary. This matches exactly the JSON dictionary in *optionSpec*, with values in place of the types.
2) *dataSource*: a string, the name of the current Galyleo Table or Galyleo View

Typically, an extension to Galyleo's charting capabilities is done by incorporating the library as described in the previous section, implementing the API given in this section, and then publishing the result as a component

Extending Galyleo's Widget Set

A widget is a graphical item used to filter data. It operates on a single column on any table in the current data set. It is either a range filter (which selects a range of numeric values) or a select filter (which selects a specific value, or a set of specific values). The API that is implemented consists only of properties.

1) *valueChanged*: a signal, which is fired whenever the value of the widget is changed
2) *value*: read-write. The current value of the widget
3) *filter*: read-only. The current filter function, as a JSON structure
4) *allValues*: read-write, select filters only.
5) *column*: read-only. The name of the column of this widget. Set when the widget is created
6) *numericSpec*: read-write. A dictionary containing the numeric specification for a numeric or date filter

Widgets are typically designed as a standard Lively graphical component, much as the slider described above.

Integration into Jupyter Lab: The Galyleo Extension

Galyleo is a standalone web application that is integrated into JupyterLab using an iframe inside a JupyterLab tab for physical design. A small JupyterLab extension was built that implements the JupyterLab editor API. The JupyterLab extension has two major functions: to handle read/write/undo requests from the JupyterLab menus and file browser, and receive and transmit messages from the running Jupyter kernels to update tables on the Dashboard Studio, and to handle the reverse messages where the studio requests data from the kernel.

Standard Jupyter and browser mechanisms are used. File system requests come to the extension from the standard Jupyter API, exactly the same requests and mechanisms that are sent to a Markdown or Notebook editor. The extension receives them, and then uses standard browser-based messaging (*window.postMessage*) to signal the standalone web app. Similarly, when the extension

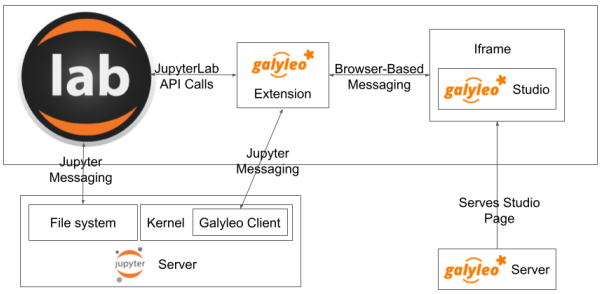


Fig. 7: *Figure 7. Galyleo Extension Architecture*

makes a request of JupyterLab, it does so through this mechanism and a receiver in the extension gets it and makes the appropriate method calls within JupyterLab to achieve the objective.

When a kernel makes a request through the Galyleo Client, this is handled exactly the same way. A Jupyter messaging server within the extension receives the message from the kernel, and then uses browser messaging to contact the application with the request, and does the reverse on a Galyleo message to the kernel.

This is a highly efficient method of interaction, since browser-based messaging is in-memory transactions on the client machine.

It's important to note that there is nothing Galyleo-specific about the extension: the Galyleo Extension is a general method for *any* standalone web editor (e.g., a slide or drawing editor) to be integrated into JupyterLab. The JupyterLab connection is a few tens of lines of code in the Galyleo Dashboard. The extension is slightly more complex, but it can be configured for a different application with a simple data structure which specifies the URL of the application, file type and extension to be manipulated, and message list.

The Jupyter Computer

The implications of the Galyleo Extension go well beyond visualization and dashboards and easy publication in JupyterLab. JupyterLab is billed as the next-generation integrated Development Environment for Jupyter, but in fact it is substantially more than that. It is the user interface and windowing system for Cloud-based personal computing. Inspired by previous extensions such as the Vega Extension, the Galyleo Extensions seeks to provide the final piece of the puzzle.

Consider a Jupyter server in the Cloud, served from a JupyterHub such as the Berkeley Data Hub. It's built from a base Ubuntu image, with the standard Jupyter libraries installed and, importantly, a UI that includes a Linux terminal interface. Any Linux executable can be installed in the Jupyter server image, as can any Jupyter kernel, and any collection of libraries. The Jupyter server has per-user persistent storage, which is organized in a standard Linux filesystem. This makes the Jupyter server a curated execution environment with a Linux command-line interface and a Notebook interface for Jupyter execution.

A JupyterHub similar to Berkeley Data Hub (essentially, anything built from Zero 2 Jupyter Hub or Q-Hub) comes with a number of "environments". The user chooses the environment on startup. Each environment comes with a built-in set of libraries and executables designed for a specific task or set of tasks. The number of environments hosted by a server is arbitrary, and the cost is only the cost of maintaining the Dockerfile for each environment.

An environment is easy to design for a specific class, project, or task; it's simply adding libraries and executables to a base Dockerfile. It must be tested, of course, but everything must be. And once it is tested, the burden of software maintenance and installation is removed from the user; the user is already in a task-customized, curated environment. Of course, the usual installation tools (*apt*, *pip*, *conda*, *easy_install*) can be pre-loaded (they're just executables) so if the environment designer missed something it can be added by the end user.

Though a user can only be in one environment at a time, persistent storage is shared across all environments, meaning switching environments is simply a question of swapping one environment out and starting another.

Viewed in this light, a JupyterHub is a multi-purpose computer in the Cloud, with an easy-to-use UI that presents through a browser. JupyterLab isn't simply an IDE; it's the window system and user interface for this computer. The JupyterLab launcher is the desktop for this computer (and it changes what's presented, depending on the environment); the file browser is the computer's file browser, and the JupyterLab API is the equivalent of the Windows or MacOS desktop APIs and window system that permits third parties to build applications for this.

This Jupyter Computer has a large number of advantages over a standard desktop or laptop computer. It can be accessed from any device, anywhere on Earth with an Internet connection. Software installation and maintenance issues are nonexistent. Data loss due to hardware failure is extremely unlikely; backups are still required to prevent accidental data loss (e.g., erroneous file deletion), but they are far easier to do in a Cloud environment. Hardware resources such as disk, RAM, and CPU can be added rapidly, on a permanent or temporary basis. Relatively exotic resources (e.g., GPUs) can also be added, again on an on-demand, temporary basis.

The advantages go still further than that. Any resource that can be accessed over a network connection can be added to the Jupyter Computer simply by adding the appropriate accessor library to an environment's Dockerfile. For example, a database solution such as Snowflake, BigQuery, or Amazon Aurora (or one of many others) can be "installed" by adding the relevant library module to the environment. Of course, the user will need to order the database service from the relevant provider, and obtain authentication tokens, and so on -- but this is far less troublesome than even maintaining the library on the desktop.

However, to date the Jupyter Computer only supports a few window-based applications, and adding a new application is a time-consuming development task. The applications supported are familiar and easy to enumerate: a Notebook editor, of course; a Markdown Viewer; a CSV Viewer; a JSON Viewer (not inline editor), and a text editor that is generally used for everything from Python files to Markdown to CSV.

This is a small subset of the rich range of JavaScript/HTML5 applications which have significant value for Jupyter Computer users. For example, the Ace Code Editor supports over 110 languages and has the functionality of popular desktop editors such as Vim and Sublime Text. There are over 1100 open-source drawing applications on the JavaScript/HTML5 platform; multiple spreadsheet applications, the most notable being jExcel, and many more.

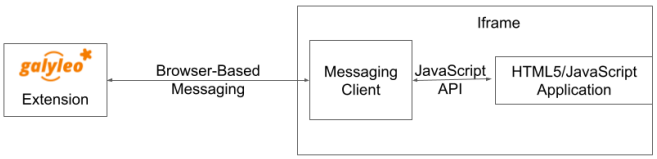


Fig. 8: *Figure 8. Galyleo Extension Application-Side messaging*

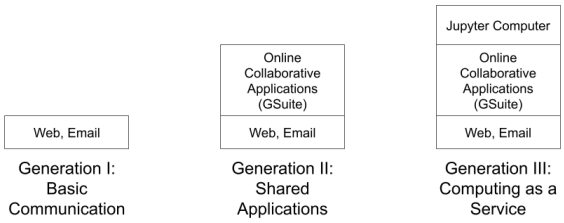


Fig. 9: *Figure 9. Generations of Internet Computing*

Up until now, adding a new application to JupyterLab involved writing a hand-coded extension in Typescript, and compiling it into JupyterLab. However, the Galyleo Extension has been designed so that any HTML5/JavaScript application can be added easily, simply by configuring the Galyleo Extension with a small JSON file.

The promise of the Galyleo Extension is that it can be adapted to *any* open-source JavaScript/HTML5 application very easily. The Galyleo Extension merely needs the:

- URL of the application
- File extension that the application reads/writes
- URL of an image for the launcher
- Name of the application for the file menu

The application must implement a small messaging client, using the standard JavaScript messaging interface, and implement the calls the Galyleo Extension makes. The conceptual picture is shown im Figure 8.

And it must support (at a minimum) messages to read and write the file being edited.

The Third Generation of Network Computing

The World-Wide Web and email comprised the first generation of Internet computing (the Internet had been around for a decade before the Web, and earlier networks dated from the sixties, but the Web and email were the first mass-market applications on the network), and they were very simple -- both were document-exchange applications, using slightly different protocols. The second generation of Network applications were the siloed productivity applications, where standard desktop applications moved to the Cloud. The most famous example is of course GSuite and Office 365, but there were and are many others -- Canva, Loom, Picasa, as well as a large number of social/chat/social media applications. What they all had in common was that they were siloed applications which, with the exception of the office suites, didn't even share a common store. In many ways, this second generation of network applications recapitulates the era immediately prior to the introduction of the personal computer. That era was dominated by single-application computers such as word processors, which were simply computers with a hardcoded program loaded into ROM.

The Word Processor era was due to technological limitations -- the processing power and memory to run multiple programs simply wasn't available on low-end hardware, and PC operating systems didn't yet exist. In some sense, the current second generation of Internet Computing suffers from similar technological constraints. The "Operating System" for Internet Computing doesn't yet exist. The Jupyter Computer can provide it.

To see the difference that this can make, consider LaTeX (perhaps preceded by Docutils, as is the case for SciPy) preparation of a document. On a personal computer, it's fairly straightforward; the user uses any of a wide variety of text editors to prepare the document, any of a wide variety of productivity and illustrator programs to prepare the images, runs this through a local sequence of commands (e.g., *pdflatex paper; bibtex paper; pdflatex paper.* Usually Github or another repository is used for storage and collaboration.

In a Cloud service, this is another matter. There is at most one editor, selected by the service, on the site. There is no image editing or illustrator program that reads and writes files on the site. Auxiliary tools, such as a bib searcher, aren't present or aren't customizable. The service has its own siloed storage, its own text editor, and its own document-preparation pipeline. The tools (aside from the core document-preparation program) are primitive. The online service has two advantages over the personal-device service. Collaboration is generally built-in, with multiple people having access to the project, and the software need not be maintained. Aside from that, the personal-device experience is generally superior. In particular, the user is free to pick their own editor, and doesn't have to orchestrate multiple downloads and uploads from various websites. The usual collection of command-line utilities are available to small touchups.

The third generation of Internet Computing represented by the Jupyter Computer. This offers a Cloud experience similar to the personal computer, but with the scalability, reliability, and ease of collaboration of the Cloud.

Conclusion and Further Work

The vision of the Jupyter Computer, bringing the power of the Cloud to the personal computing experience has been started with Galyleo. It will not end there. At the heart of it is a composition of two broadly popular platforms: HTML5/JavaScript for presentation and interaction, and the various Jupyter kernels for server-side analytics. Galyleo is a start at seamless interaction of these two platforms. Continuing and extending this is further development of narrow-waist protocols to permit maximal independent development and extension.

Acknowledgements

The authors wish to thank Alex Yang, Diptorup Deb, and for their insightful comments, and Meghann Agarwal for stewardship. We have received invaluable help from Robert Krahn, Marko Röder, Jens Lincke and Linus Hagemann. We thank the engageLively team for all of their support and help: Tim Braman, Patrick Scaglia, Leighton Smith, Sharon Zehavi, Igor Zhukovsky, Deepak Gupta, Steve King, Rick Rasmussen, Patrick McCue, Jeff Wade, Tim Gibson. The JupyterLab development community has been helpful and supportive; we want to thank Tony Fast, Jason Grout, Mehmet Bektas, Isabela Presedo-Floyd, Brian

Granger, and Michal Krassowski. The engageLively Technology Advisory Board has helped shape these ideas: Ani Mardurkar, Priya Joseph, David Peterson, Sunil Joshi, Michael Czahor, Isha Oke, Petrus Zwart, Larry Rowe, Glenn Ricart, Sunil Joshi, Antony Ng. We want to thank the people from the AWS team that have helped us tremendously: Matt Vail, Omar Valle, Pat Santora. Galyleo has been dramatically improved with the assistance of our Japanese colleagues at KCT and Pacific Rim Technologies: Yoshio Nakamura, Ted Okasaki, Ryder Saint, Yoshikazu Tokushige, and Naoyuki Shimazaki. Our undestanding of Jupyter in an academic context came from our colleagues and friends at Berkeley, the University of Victoria, and UBC: Shawna Dark, Hausi Müller, Ulrike Stege, James Colliander, Chris Holdgraf, Nitesh Mor. Use of Jupyter in a research context was emphasized by Andrew Weidlea, Eli Dart, Jeff D'Ambrogia. We benefitted enormously from the CITRIS Foundry: Alic Chen, Jing Ge, Peter Minor, Kyle Clark, Julie Sammons, Kira Gardner. The Alchemist Accelerator was central to making this product: Ravi Belani, Arianna Haider, Jasmine Sunga, Mia Scott, Kenn So, Aaron Kalb, Adam Frankl. Kris Singh was a constant source of inspiration and help. Larry Singer gave us tremendous help early on. Vibhu Mittal more than anyone inspired us to pursue this road. Ken Lutz has been a constant sounding board and inspiration, and worked hand-in-hand with us to develop this product. Our early customers and partners have been and continue to be a source of inspiration, support, and experience that is absolutely invaluable: Jonathan Tan, Roger Basu, Jason Koeller, Steve Schwab, Michael Collins, Alefiya Hussain, Geoff Lawler, Jim Chimiak, Fraukë Tillman, Andy Bavier, Andy Milburn, Augustine Bui. All of our customers are really partners, none moreso than the fantastic teams at Tanjo AI and Ultisim: Bjorn Nordwall, Ken Lane, Jay Sanders, Eric Smith, Miguel Matos, Linda Bernard, Kevin Clark, and Richard Boyd. We want to especially thank our investors, who bet on this technology and company.

REFERENCES

[ABF20] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. URL: https://doi.org/10.1145/3428290, doi:10.1145/3428290.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, dec 2011. URL: https://doi.org/10.1109/TVCG.2011.185, doi:10.1109/TVCG.2011.185.

[cha] Chart.js. URL: https://www.chartjs.org/.

[Cro06] D. Crockford. The application/json media type for javascript object notation (json). RFC 4627, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4627.txt. URL: http://www.rfc-editor.org/rfc/rfc4627.txt, doi:10.17487/rfc4627.

[Dev] Erik Devaney. How to create a pivot table in excel: A step-by-step tutorial. URL: https://blog.hubspot.com/marketing/how-to-create-pivot-table-tutorial-ht.

[DGHP13] Marcello D'Agostino, Dov M Gabbay, Reiner Hähnle, and Joachim Posegga. *Handbook of tableau methods*. Springer Science & Business Media, 2013.

[goo] Charts: google developers. URL: https://developers.google.com/chart/.

[HIK+16] Matthew Hemmings, Daniel Ingalls, Robert Krahn, Rick McGeer, Glenn Ricart, Marko Röder, and Ulrike Stege. Livetalk: A framework for collaborative browser-based replicated-computation applications. In *2016 28th International Teletraffic Congress (ITC 28)*, volume 01, pages 270–277, 2016. doi:10.1109/ITC-28.2016.144.

[hol22a] High-level tools to simplify visualization in python, Apr 2022. URL: https://holoviz.org/.

[hol22b] Installation - holoviews v1.14.9, May 2022. URL: https://holoviews.org/.

[IFH+16] Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. A world of active objects for work and play: The first ten years of lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, page 238–249, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2986012.2986029, doi:10.1145/2986012.2986029.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 318–326, New York, NY, USA, 1997. Association for Computing Machinery. URL: https://doi.org/10.1145/263698.263754, doi:10.1145/263698.263754.

[IPU+08] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The lively kernel a self-supporting system on a web page. In *Workshop on Self-sustaining Systems*, pages 31–50. Springer, 2008. doi:10.1007/978-3-540-89275-5_2.

[jup] Jupyterlab documentation. URL: https://jupyterlab.readthedocs.io/en/stable/.

[KIH+09] Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. Lively wiki a development environment for creating and sharing active web content. In *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, WikiSym '09, New York, NY, USA, 2009. Association for Computing Machinery. URL: https://doi.org/10.1145/1641309.1641324, doi:10.1145/1641309.1641324.

[KRB18] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: Insights from the practice. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1090–1101, New York, NY, USA, 2018. Association for Computing Machinery. URL: https://doi.org/10.1145/3180155.3180200, doi:10.1145/3180155.3180200.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. *Jupyter Notebooks - a publishing format for reproducible computational workflows*. IOS Press, 2016. URL: https://eprints.soton.ac.uk/403913/.

[lea] An open-source javascript library for interactive maps. URL: https://leafletjs.com/.

[LKI+12] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. The lively partsbin–a cloud-based repository for collaborative development of active web content. In *2012 45th Hawaii International Conference on System Sciences*, pages 693–701, 2012. doi:10.1109/HICSS.2012.42.

[loo] Looker. URL: https://looker.com/.

[LS10] Bruce Lawson and Remy Sharp. *Introducing HTML5*. New Riders Publishing, USA, 1st edition, 2010.

[mdn] Window.postmessage() - web apis: Mdn. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[MRR+10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010. URL: https://doi.org/10.1145/1868358.1868363, doi:10.1145/1868358.1868363.

[MS95] John H Maloney and Randall B Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, 1995. URL: https://doi.org/10.1145/215585.215636, doi:10.1145/215585.215636.

[ope] Openlayers. URL: https://openlayers.org/.

[pan22] Panel, May 2022. URL: https://panel.holoviz.org/.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[plo] Dash overview. URL: https://plotly.com/dash/.

[SKH21] Robin Schrieber, Robert Krahn, and Linus Hagemann. lively.next, 2021.

[TM17] Antero Taivalsaari and Tommi Mikkonen. The web as a software platform: Ten years later. In *International Conference on Web Information Systems and Technologies*, volume 2, pages 41–50. SCITEPRESS, 2017. doi:10.5220/0006234800410050.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. volume 22, page 227–242, New York, NY, USA, dec 1987. Association for Computing Machinery. URL: https://doi.org/10.1145/38807.38828, doi:10.1145/38807.38828.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a.

USACE Coastal Engineering Toolkit and a Method of Creating a Web-Based Application

Amanda Catlett‡*, Theresa R. Coumbe‡, Scott D. Christensen‡, Mary A. Byrant‡

Abstract—In the early 1990s the Automated Coastal Engineering Systems, ACES, was created with the goal of providing state-of-the-art computer-based tools to increase the accuracy, reliability, and cost-effectiveness of Corps coastal engineering endeavors. Over the past 30 years, ACES has become less and less accessible to engineers. An updated version of ACES was necessary for use in coastal engineering. Our goal was to bring the tools in ACES to a user-friendly web-based dashboard that would allow a wide range of users to be able to easily and quickly visualize results. We will discuss how we restructured the code using class inheritance and the three libraries Param, Panel, and HoloViews to create an extensible, interactive, graphical user interface. We have created the USACE Coastal Engineering Toolkit, UCET, which is a web-based application that contains 20 of the tools in ACES. UCET serves as an outline for the process of taking a model or set of tools and developing web-based application that can produce visualizations of the results.

Index Terms—GUI, Param, Panel, HoloViews

Introduction

The Automated Coastal Engineering System (ACES) was developed in response to the charge by the LTG E. R. Heiberg III, who was the Chief of Engineers at the time, to provide improved design capabilities to the Corps coastal specialists. [Leenknecht] In 1992, ACES was presented as an interactive computer-based design and analysis system in the field of coastal engineering. The tools consist of seven functional areas which are: Wave Prediction, Wave Theory, Structural Design, Wave Runup Transmission and Overtopping, Littoral Process, and Inlet Processes. These functional areas contain classical theory describing wave motion, to expressions resulting from tests of structures in wave flumes, and numerical models describing the exchange of energy from the atmosphere to the sea surface. The math behind these uses anything from simple algebraic expressions, both theoretical and empirical, to numerically intense algorithms. [Leenknecht][UG][shankar]

Originally, ACES was written in FORTRAN 77 resulting in a decreased ability to use the tool as technology has evolved. In 2017, the codebase was converted from FORTRAN 77 to MATLAB and Python. This conversion ensured that coastal engineers using this tool base would not need training in yet another coding language. In 2020, the Engineered Resilient Systems (ERS) Rapid Application Development (RAD) team undertook the project with the goal of deploying the ACES tools as a web-based application, and ultimately renamed it to: USACE Coastal Engineering Toolkit (UCET).

The RAD team focused on updating the Python codebase utilizing Python's object-oriented programming and the newly developed HoloViz ecosystem. The team refactored the code to implement inheritance so the code is clean, readable, and scalable. The tools were expanded to a Graphical User Interface (GUI) so the implementation to a web-app would provide a user-friendly experience. This was done by using the HoloViz-maintained libraries: Param, Panel, and Holoviews.

This paper will discuss some of the steps that were taken by the RAD team to update the Python codebase to create a panel application of the coastal engineering tools. In particular, refactoring the input and output variables with the Param library, the class hierarchy used, and utilization of Panel and HoloViews for a user-friendly experience.

Refactoring Using Param

Each coastal tool in UCET has two classes, the model class and the GUI class. The model class holds input and output variables and the methods needed to run the model. Whereas the GUI class holds information for GUI visualization. To make implementation of the GUI more seamless we refactored model variables to utilize the Param library. Param is a library that has the goal of simplifying the codebase by letting the programmer explicitly declare the types and values of parameters accepted by the code. Param can also be seamlessly used when implementing the GUI through Panel and HoloViews.

Each UCET tool's model class declares the input and output values used in the model as class parameters. Each input and output variables are declared and given the following metadata features:

- **default**: each input variable is defined as a Param with a default value defined from the 1992 ACES user manual
- **bounds**: each input variable is defined with range values defined in the 1992 ACES user manual
- **doc or docstrings**: input and output variables have the expected variable and description of the variable defined as a doc. This is used as a label over the input and output widgets. Most docstrings follow the pattern of <variable>:<description of variable [units, if any]>
- **constant**: the output variables all set constant equal True, thereby restricting the user's ability to manipulate the

* *Corresponding author: amanda.r.catlett@erdc.dren.mil*
‡ *ERDC*

value. Note that when calculations are being done they will need to be inside a with param.edit_constant(self) function

- **precedence**: input and output variables will use precedence when there are instances where the variable does not need to be seen.

The following is an example of an input parameter:

```
H = param.Number(
    doc='H: wave height [{distance_unit}]',
    default=6.3,
    bounds=(0.1, 200)
)
```

An example of an output variable is:

```
L = param.Number(
    doc='L: Wavelength [{distance_unit}]',
    constant=True
)
```

The model's main calculation functions mostly remained unchanged. However, the use of Param eliminated the need for code that handled type checking and bounds checks.

Class Hierarchy

UCET has twenty tools from six of the original seven functional areas of ACES. When we designed our class hierarchy, we focused on the visualization of the web application rather than functional areas. Thus, each tool's class can be categorized into Base-Tool, Graph-Tool, Water-Tool, or Graph-Water-Tool. The Base-Tool has the coastal engineering models that do not have any water property inputs (such as water density) in the calculations and no graphical output. The Graph-Tool has the coastal engineering models that do not have any water property inputs in the calculations but have a graphical output. Water-Tool has the coastal engineering models that have water property inputs in the calculations and no graphical output. Graph-Water-Tool has the coastal engineering models that have water property inputs in the calculations and has a graphical output. Figure 1 shows a flow of inheritance for each of those classes.

There are two types of general categories for the classes in the UCET codebase: utility and tool-specific. Utility classes have methods and functions that are utilized across more than one tool. The Utility classes are:

- **BaseDriver**: holds methods and functions that each tool needs to collect data, run coastal engineering models, and print data.
- **WaterDriver**: has the methods that make water density and water weight available to the models that need those inputs for the calculations.
- **BaseGui**: has the functions and methods for the visualization and utilization of all inputs and outputs within each tool's GUI.
- **WaterTypeGui**: has the widget for water selection.
- **TabulatorDataGui**: holds the functions and methods used for visualizing plots and the ability to download the data that is used for plotting.

Each coastal tool in UCET has two classes, the model class and the GUI class. The model class holds input and output variables and the methods needed to run the model. The model class either directly inherits from the BaseDriver or the WaterTypeDriver. The tool's GUI class holds information for GUI visualization that is different from the BaseGui, WaterTypeGUI, and TabulatorDataGui classes. In figure 1 the model classes are labeled as: Base-Tool Class, Graph-Tool Class, Water-Tool Class, and Graph-Water-Tool Class and each has a corresponding GUI class.

Due to the inheritance in UCET, the first two questions that can be asked when adding a tool are: 'Does this tool need water variables for the calculation?' and 'Does this tool have a graph?'. The developer can then add a model class and a GUI class and inherit based on figure 1. For instance, Linear Wave Theory is an application that yields first-order approximations for various parameters of wave motion as predicted by the wave theory. It provides common items of interest such as water surface elevation, general wave properties, particle kinematics and pressure as a function of wave height and period, water depth, and position in the wave form. This tool uses water density and has multiple graphs in its output. Therefore, Linear Wave Theory is considered a Graph-Water-Tool and the model class will inherit from WaterTypeDriver and the GUI class will inherit the linear wave theory model class, WaterTypeGui, and TabularDataGui.

GUI Implementation Using Panel and HoloViews

Each UCET tool has a GUI class where the Panel and HoloView libraries are implemented. Panel is a hierarchical container that can layout panes, widgets, or other Panels in an arrangement that forms an app or dashboard. The Pane is used to render any widget-like object such as Spinner, Tabulator, Buttons, CheckBox, Indicators, etc. Those widgets are used to gather user input and run the specific tool's model.

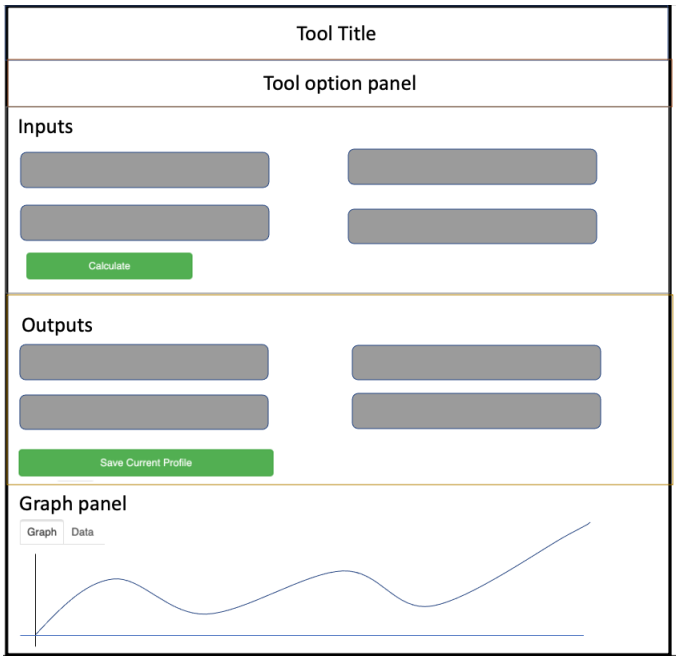
UCET utilizes the following widgets to gather user input:

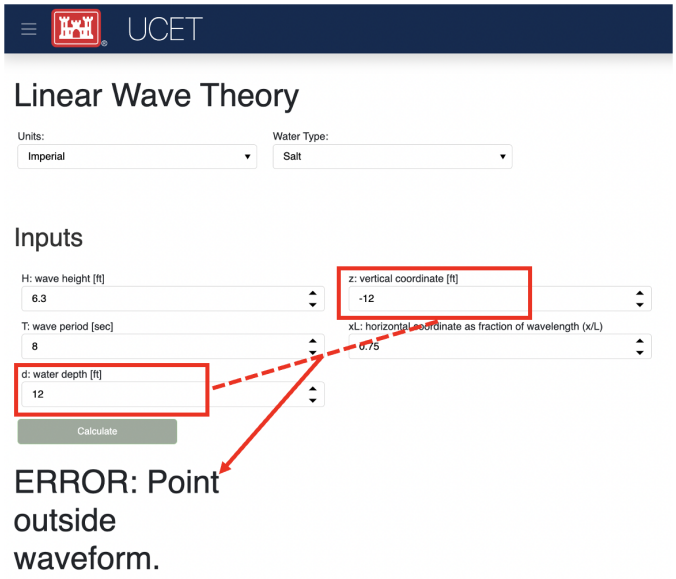
- **Spinner**: single numeric input values
- **Tabulator**: table input data
- **CheckBox**: true or false values
- **Drop down**: items that have a list of pre-selected values, such as which units to use

UCET utilizes indicators.Number, Tabulator, and graphs to visualize the outputs of the coastal engineering models. A single number is shown using indicators.Number and graph data is displayed using the Tabulator widget to show the data of the graph. The graphs are created using HoloViews and have tool options such as pan, zooming, and saving. Buttons are used to calculate, save the current run, and save the graph data.

All of these widgets are organized into 5 panels: title, options, inputs, outputs, and graph. The BaseGui/WaterTypeGui/TabularDataGui have methods that organize the widgets within the 5 panels that most tools follow. The "options" panel has a row that holds the dropdown selections for units and water type (if the tool is a Water-Tool). Some tools have a second row in the "options" panel with other drop-down options. The input panel has two columns for spinner widgets with a calculation button at the bottom left. The output panel has two columns of indicators.Number for the single numeric output values. At the bottom of the output panel there is a button to "save the current profile". The graph panel is tabbed where the first tab shows the graph and the second tab shows the data provided within the graph. An visual outline of this can ben seen in the following figure. Some of the UCET tools have more complicated input or output visualizations and that tool's GUI class will add or modify methods to meet the needs of that tool.

The general outline of a UCET tool for the GUI.





Current State

UCET approaches software development from the perspective of someone within the field of Research and Development. Each tool within UCET is not inherently complex from the traditional software perspective. However, this codebase enables researchers to execute complex coastal engineering models in a user-friendly environment by leveraging open-source libraries in the scientific Python ecosystem such as: Param, Panel, and HoloViews.

Currently, UCET is only deployed using a command line interface panel serve command. UCET is awaiting the Security Technical Implementation Guide process before it can be launched as a website. As part of this security vetting process we plan to leverage continuous integration/continuous development (CI/CD) tools to automate the deployment process. While this process is happening, we have started to get feedback from coastal engineers to update the tools usability, accuracy, and adding suggested features. To minimize the amount of computer science knowledge the coastal engineers need, our team created a batch script. This script creates a conda environment, activates and runs the panel serve command to launch the app on a local host. The user only needs to click on the batch script for this to take place.

Other tests are being created to ensure the accuracy of the tools using a testing framework to compare output from UCET with that of the FORTRAN original code. The biggest barrier to this testing strategy is getting data from the FORTRAN to compare with Python. Currently, there are tests for most of the tools that read a CSV file of input and output results from FORTRAN and compare with what the Python code is calculating.

Our team has also compiled an updated user guide on how to use the tool, what to expect from the tool, and a deeper description on any warning messages that might appear as the user adds input values. An example of a warning message would be, if a user chooses input values that make it so the application does not make physical sense, a warning message will appear under the output header and replace all output values. For a more concrete example: Linear Wave Theory has a vertical coordinate (z) and the water depth (d) as input values and when those values sum is less than zero the point is outside the waveform. Therefore, if a user makes a combination where the sum is less than zero, UCET will post a warning to tell the user that the point is outside the waveform. See the below figure for an example The developers have been documenting this project using GitHub and JIRA.

An example of a warning message based on chosen inputs.

Results

Linear Wave Theory was described in the class hierarchy example. This Graph-Water-Tool utilizes most of the BaseGui methods. The biggest difference is instead of having three graphs in the graph panel there is a plot selector drop down where the user can select which graph they want to see.

Windspeed Adjustment and Wave Growth provides a quick and simple estimate for wave growth over open-water and restricted fetches in deep and shallow water. This is a Base-Tool as there are no graphs and no water variables for the calculations. This tool has four additional options in the options panel where the user can select the wind observation type, fetch type, wave equation type, and if knots are being used. Based on the selection of these options, the input and output variables will change so only what is used or calculated for those selections are seen.

Conclusion and Future Work

Thirty years ago, ACES was developed to provide improved design capabilities to Corps coastal specialists and while these tools are still used today, it became more and more difficult for users to access them. Five years ago, there was a push to update the code base to one that coastal specialists would be more familiar with: MATLAB and Python. Within the last two years the RAD team was able to finalize the update so that the user can access these tools without having years of programming experience. We were able to do this by utilizing classes, inheritance, and the Param, Panel, and HoloViews libraries. The use of inheritance has allowed for shorter code-bases and also has made it so new tools can be added to the toolkit. Param, Panel, and HoloViews work cohesively together to not only run the models but make a simple interface.

Future work will involve expanding UCET to include current coastal engineering models, and completing the security vetting

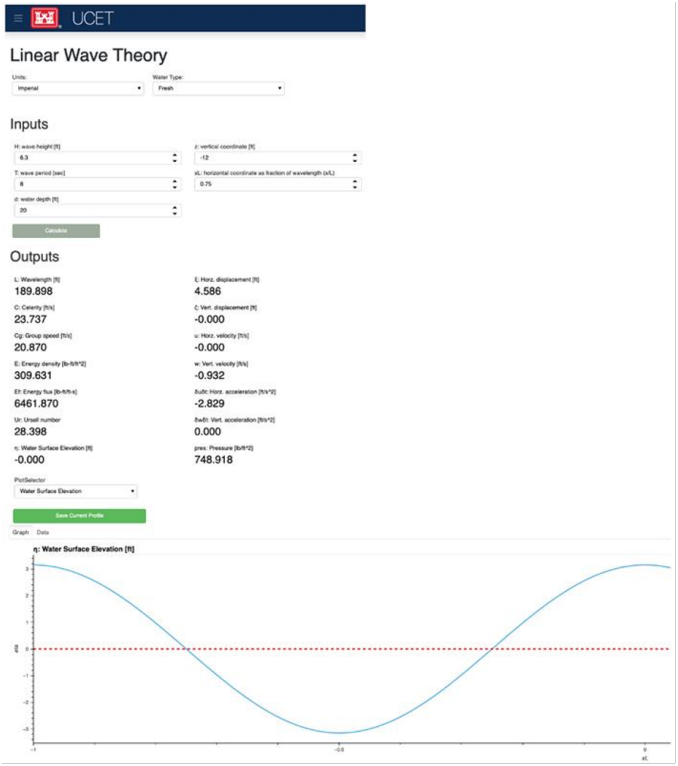


Fig. 1: *Screen shot of Linear Wave Theory*

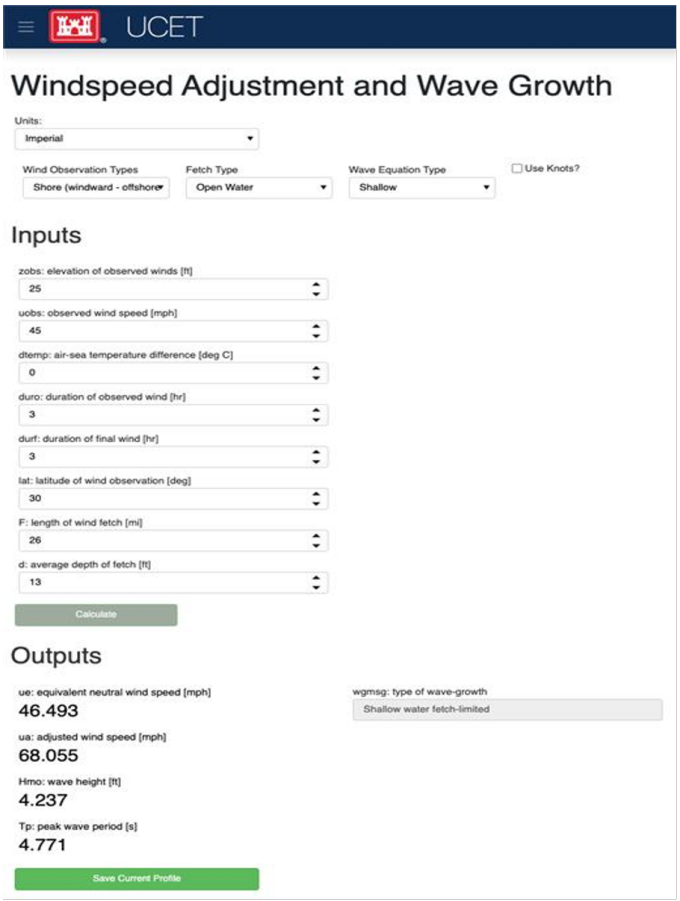


Fig. 2: *Screen shot of Windspeed Adjustment and Wave Growth*

process to deploy to a publicly accessible website. We plan to incorporate an automated CI/CD to ensure smooth deployment of future versions. We also will continue to incorporate feedback from users and refine the code to ensure the application provides a quality user experience.

REFERENCES

[Leenknecht] David A. Leenknecht, Andre Szuwalski, and Ann R. Sherlock. 1992. Automated Coastal Engineering System -Technical Reference. Technical report. https://usace.contentdm.oclc.org/digital/collection/p266001coll1/id/2321/

[panel] "Panel: A High-Level App and Dashboarding Solution for Python." Panel 0.12.6 Documentation, Panel Contributors, 2019, https://panel.holoviz.org/.

[holoviz] "High-Level Tools to Simplify Visualization in Python." HoloViz 0.13.0 Documentation, HoloViz Authors, 2017, https://holoviz.org.

[UG] David A. Leenknecht, et al. "Automated Tools for Coastal Engineering." Journal of Coastal Research, vol. 11, no. 4, Coastal Education & Research Foundation, Inc., 1995, pp. 1108-24. https://usace.contentdm.oclc.org/digital/collection/p266001coll1/id/2321/

[shankar] N.J. Shankar, M.P.R. Jayaratne, Wave run-up and overtopping on smooth and rough slopes of coastal structures, Ocean Engineering, Volume 30, Issue 2, 2003, Pages 221-238, ISSN 0029-8018, https://doi.org/10.1016/S0029-8018(02)00016-1

Search for Extraterrestrial Intelligence: GPU Accelerated TurboSETI

Luigi Cruz‡*, Wael Farah‡, Richard Elkins‡

Abstract—A common technique adopted by the Search For Extraterrestrial Intelligence (SETI) community is monitoring electromagnetic radiation for signs of extraterrestrial technosignatures using ground-based radio observatories. The analysis is made using a Python-based software called TurboSETI to detect narrowband drifting signals inside the recordings that can mean a technosignature. The data stream generated by a telescope can easily reach the rate of terabits per second. Our goal was to improve the processing speeds by writing a GPU-accelerated backend in addition to the original CPU-based implementation of the de-doppler algorithm used to integrate the power of drifting signals. We discuss how we ported a CPU-only program to leverage the parallel capabilities of a GPU using CuPy, Numba, and custom CUDA kernels. The accelerated backend reached a speed-up of an order of magnitude over the CPU implementation.

Index Terms—gpu, numba, cupy, seti, turboseti

1. Introduction

The Search for Extraterrestrial Intelligence (SETI) is a broad term utilized to describe the effort of locating any scientific proof of past or present technology that originated beyond the bounds of Earth. SETI can be performed in a plethora of ways: either actively by deploying orbiters and rovers around planets/moons within the solar system, or passively by either searching for biosignatures in exoplanet atmospheres or "listening" to technologically-capable extraterrestrial civilizations. One of the most common techniques adopted by the SETI community is monitoring electromagnetic radiation for narrowband signs of technosignatures using ground-based radio observatories. This search can be performed in multiple ways: equipment primarily built for this task, like the Allen Telescope Array (California, USA), renting observation time, or in the background while the primary user is conducting other observations. Other radio-observatories useful for this search include the MeerKAT Telescope (Northern Cape, South Africa), Green Bank Telescope (West Virginia, USA), and the Parkes Telescope (New South Wales, Australia). The operation of a radio-telescope is similar to an optical telescope. Instead of using optics to concentrate light into an optical sensor, a radio-telescope operates by concentrating electromagnetic waves into an antenna using a large reflective structure called a "dish" ([Reb82]). The interaction between the metallic antenna and the electromagnetic wave generates a faint electrical current. This effect is then quantized by an analog-to-digital converter as voltages and transmitted to a processing logic to extract useful information from it. The data stream generated by a radio telescope can easily reach the rate of terabits per second because of the ultra-wide bandwidth radio spectrum. The current workflow utilized by the Breakthrough Listen, the largest scientific research program aimed at finding evidence of extraterrestrial intelligence, consists in pre-processing and storing the incoming data as frequency-time binary files ([LCS+19]) in persistent storage for later analysis. This post-analysis is made possible using a Python-based software called TurboSETI ([ESF+17]) to detect narrowband signals that could be drifting in frequency owing to the relative radial velocity between the observer on earth, and the transmitter. The offline processing speed of TurboSETI is directly related to the scientific output of an observation. Each voltage file ingested by TurboSETI is often on the order of a few hundreds of gigabytes. To process data efficiently without Python overhead, the program uses Numpy for near machine-level performance. To measure a potential signal's drift rate, TurboSETI uses a de-doppler algorithm to align the frequency axis according to a pre-set drift rate. Another algorithm called "hitsearch" ([ESF+17]) is then utilized to identify any signal present in the recorded spectrum. These two algorithms are the most resource-hungry elements of the pipeline consuming almost 90% of the running time.

2. Approach

Multiple methods were utilized in this effort to write a GPU-accelerated backend and optimize the CPU implementation of TurboSETI. In this section, we enumerate all three main methods.

2.1. CuPy

The original implementation of TurboSETI heavily depends on Numpy ([HMvdW+20]) for data processing. To keep the number of modifications as low as possible, we implemented the GPU-accelerated backend using CuPy ([OUN+17]). This open-source library offers GPU acceleration backed by NVIDIA CUDA and AMD ROCm while using a Numpy style API. This enabled us to reuse most of the code between the CPU and GPU-based implementations.

2.1. Numba

Some computationally heavy methods of the original CPU-based implementation of TurboSETI were written in Cython. This approach has disadvantages: the developer has to be familiar with Cython syntax to alter the code; the code requires additional logic

* *Corresponding author: lfcruz@seti.org*
‡ *SETI Institute*

Double-Precision (float64)				
Impl.	Device	File A	File B	File C
Cython	CPU	0.44 min	25.26 min	23.06 min
Numba	CPU	0.36 min	20.67 min	22.44 min
CuPy	GPU	0.05 min	2.73 min	3.40 min

TABLE 1
Double precision processing time benchmark with Cython, Numba and CuPy implementation.

Single-Precision (float32)				
Impl.	Device	File A	File B	File C
Numba	CPU	0.26 min	16.13 min	16.15 min
CuPy	GPU	0.03 min	1.52 min	2.14 min

TABLE 2
Single precision processing time benchmark with Numba and CuPy implementation.

to be compiled at installation time. Consequently, it was decided to replace Cython with pure Python methods decorated with the Numba ([LPS15]) accelerator. By leveraging the power of the Just-In-Time (JIT) compiler from Low Level Virtual Machine (LLVM), Numba can compile Python code into assembly code as well as apply Single Instruction/Multiple Data (SIMD) acceleration instructions to achieve near machine-level speeds.

2.2. Single-Precision Floating-Point

The original implementation of the software handled the input data as double-precision floating-point numbers. This behavior would cause all the mathematical operations to take significantly longer to process because of the extended precision. The ultimate precision of the output product is inherently limited by the precision of the original input data which in most cases is represented by an 8-bit signed integer. Therefore, the addition of a single-precision floating-point number decreased the processing time without compromising the useful precision of the output data.

3. Results

To test the speed improvements between implementations we used files from previous observations coming from different observatories. Table 1 indicates the processing times it took to process three different files in double-precision mode. We can notice that the CPU implementation based on Numba is measurably faster than the original CPU implementation based on Cython. At the same time, the GPU-accelerated backend processed the data from 6.8 to 9.3 times faster than the original CPU-based implementation.

Table 2 indicates the same results as Table 1 but with single-precision floating points. The original Cython implementation was left out because it doesn't support single-precision mode. Here, the same data was processed from 7.5 to 10.6 times faster than the Numba CPU-based implementation.

To illustrate the processing time improvement, a single observation containing 105 GB of data was processed in 12 hours by the original CPU-based TurboSETI implementation on an i7-7700K Intel CPU, and just 1 hour and 45 minutes by the GPU-accelerated backend on a GTX 1070 Ti NVIDIA GPU.

4. Conclusion

The original implementation of TurboSETI worked exclusively on the CPU to process data. We implemented a GPU-accelerated backend to leverage the massive parallelization capabilities of a graphical device. The benchmark performed shows that the new CPU and GPU implementation takes significantly less time to process observation data resulting in more science being produced. Based on the results, the recommended configuration to run the program is with single-precision calculations on a GPU device.

REFERENCES

[ESF+17] J. Emilio Enriquez, Andrew Siemion, Griffin Foster, Vishal Gajjar, Greg Hellbourg, Jack Hickish, Howard Isaacson, Danny C. Price, Steve Croft, David DeBoer, Matt Lebofsky, David H. E. MacMahon, and Dan Werthimer. The breakthrough listen search for intelligent life: 1.1–1.9 ghz observations of 692 nearby stars. *The Astrophysical Journal*, 849(2):104, Nov 2017. URL: https://ui.adsabs.harvard.edu/abs/2017ApJ...849..104E/abstract, doi:10.3847/1538-4357/aa8d1b.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[LCS+19] Matthew Lebofsky, Steve Croft, Andrew P. V. Siemion, Danny C. Price, J. Emilio Enriquez, Howard Isaacson, David H. E. MacMahon, David Anderson, Bryan Brzycki, Jeff Cobb, Daniel Czech, David DeBoer, Julia DeMarines, Jamie Drew, Griffin Foster, Vishal Gajjar, Nectaria Gizani, Greg Hellbourg, Eric J. Korpela, and Brian Lacki. The breakthrough listen search for intelligent life: Public data, formats, reduction, and archiving. *Publications of the Astronomical Society of the Pacific*, 131(1006):124505, Nov 2019. URL: https://arxiv.org/abs/1906.07391, doi:10.1088/1538-3873/ab3e82.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[OUN+17] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[Reb82] Grote Reber. *Cosmic Static*, pages 61–69. Springer Netherlands, Dordrecht, 1982. URL: https://doi.org/10.1007/978-94-009-7752-5_6, doi:10.1007/978-94-009-7752-5_6.

Experience report of physics-informed neural networks in fluid simulations: pitfalls and frustration

Pi-Yueh Chuang‡*, Lorena A. Barba‡

Abstract—Though PINNs (physics-informed neural networks) are now deemed as a complement to traditional CFD (computational fluid dynamics) solvers rather than a replacement, their ability to solve the Navier-Stokes equations without given data is still of great interest. This report presents our not-so-successful experiments of solving the Navier-Stokes equations with PINN as a replacement to traditional solvers. We aim to, with our experiments, prepare readers for the challenges they may face if they are interested in data-free PINN. In this work, we used two standard flow problems: 2D Taylor-Green vortex at $Re = 100$ and 2D cylinder flow at $Re = 200$. The PINN method solved the 2D Taylor-Green vortex problem with acceptable results, and we used this flow as an accuracy and performance benchmark. About 32 hours of training were required for the PINN method's accuracy to match the accuracy of a 16×16 finite-difference simulation, which took less than 20 seconds. The 2D cylinder flow, on the other hand, did not produce a physical solution. The PINN method behaved like a steady-flow solver and did not capture the vortex shedding phenomenon. By sharing our experience, we would like to emphasize that the PINN method is still a work-in-progress, especially in terms of solving flow problems without any given data. More work is needed to make PINN feasible for real-world problems in such applications. (Reproducibility package: [Chu22].)

Index Terms—computational fluid dynamics, deep learning, physics-informed neural network

1. Introduction

Recent advances in computing and programming techniques have motivated practitioners to revisit deep learning applications in computational fluid dynamics (CFD). We use the verb "revisit" because deep learning applications in CFD already existed going back to at least the 1990s, for example, using neural networks as surrogate models ([LS], [FS]). Another example is the work of Lagaris and his/her colleagues ([LLF]) on solving partial differential equations with fully-connected neural networks back in 1998. Similar work with radial basis function networks can be found in reference [LLQH]. Nevertheless, deep learning applications in CFD did not get much attention until this decade, thanks to modern computing technology, including GPUs, cloud computing, high-level libraries like PyTorch and TensorFlow, and their Python APIs.

Solving partial differential equations with deep learning is particularly interesting to CFD researchers and practitioners. The PINN (physics-informed neural network) method denotes an approach to incorporate deep learning in CFD applications, where solving partial differential equations plays the key role. These partial differential equations include the well-known Navier-Stokes equations—one of the Millennium Prize Problems. The universal approximation theorem ([Hor]) implies that neural networks can model the solution to the Navier-Stokes equations with high fidelity and capture complicated flow details as long as networks are big enough. The idea of PINN methods can be traced back to [DPT], while the name PINN was coined in [RPK]. Human-provided data are not necessary in applying PINN [LMMK], making it a potential alternative to traditional CFD solvers. Sometimes it is branded as unsupervised learning—it does not rely on human-provided data, making it sound very *"AI."* It is now common to see headlines like *"AI has cracked the Navier-Stokes equations"* in recent popular science articles ([Hao]).

Though data-free PINN as an alternative to traditional CFD solvers may sound attractive, PINN can also be used under data-driven configurations, for which it is better suited. Cai et al. [CMW+] state that PINN is not meant to be a replacement of existing CFD solvers due to its inferior accuracy and efficiency. The most useful applications of PINN should be those with some given data, and thus the models are trained against the data. For example, when we have experimental measurements or partial simulation results (coarse-grid data, limited numbers of snapshots, etc.) from traditional CFD solvers, PINN may be useful to reconstruct the flow or to be a surrogate model.

Nevertheless, data-free PINN may offer some advantages over traditional solvers, and using data-free PINN to replace traditional solvers is still of great interest to researchers (e.g., [KDYI]). First, it is a mesh-free scheme, which benefits engineering problems where fluid flows interact with objects of complicated geometries. Simulating these fluid flows with traditional numerical methods usually requires high-quality unstructured meshes with time-consuming human intervention in the pre-processing stage before actual simulations. The second benefit of PINN is that the trained models approximate the governing equations' general solutions, meaning there is no need to solve the equations repeatedly for different flow parameters. For example, a flow model taking boundary velocity profiles as its input arguments can predict flows under different boundary velocity profiles after training. Conventional numerical methods, on the contrary, require repeated simulations, each one covering one boundary velocity profile. This feature could help in situations like engineering design optimization: the process of running sets of experiments to conduct parameter sweeps and find the optimal values or geometries for

* *Corresponding author: pychuang@gwu.edu*
‡ *Department of Mechanical and Aerospace Engineering, The George Washington University, Washington, DC 20052, USA*

products. Given these benefits, researchers continue studying and improving the usability of data-free PINN (e.g., [WYP], [DZ], [WTP], [SS]).

Data-free PINN, however, is not ready nor meant to replace traditional CFD solvers. This claim may be obvious to researchers experienced in PINN, but it may not be clear to others, especially to CFD end-users without ample expertise in numerical methods. Even in literature that aims to improve PINN, it's common to see only the success stories with simple CFD problems. Important information concerning the feasibility of PINN in practical and real-world applications is often missing from these success stories. For example, few reports discuss the required computing resources, the computational cost of training, the convergence properties, or the error analysis of PINN. PINN suffers from performance and solvability issues due to the need for high-order automatic differentiation and multi-objective nonlinear optimization. Evaluating high-order derivatives using automatic differentiation increases the computational graphs of neural networks. And multi-objective optimization, which reduces all the residuals of the differential equations, initial conditions, and boundary conditions, makes the training difficult to converge to small-enough loss values. Fluid flows are sensitive nonlinear dynamical systems in which a small change or error in inputs may produce a very different flow field. So to get correct solutions, the optimization in PINN needs to minimize the loss to values very close to zero, further compromising the method's solvability and performance.

This paper reports on our not-so-successful PINN story as a lesson learned to readers, so they can be aware of the challenges they may face if they consider using data-free PINN in real-world applications. Our story includes two computational experiments as case studies to benchmark the PINN method's accuracy and computational performance. The first case study is a Taylor-Green vortex, solved successfully though not to our complete satisfaction. We will discuss the performance of PINN using this case study. The second case study, flow over a cylinder, did not even result in a physical solution. We will discuss the frustration we encountered with PINN in this case study.

We built our PINN solver with the help of NVIDIA's Modulus library ([noa]). Modulus is a high-level Python package built on top of PyTorch that helps users develop PINN-based differential equation solvers. Also, in each case study, we also carried out simulations with our CFD solver, PetIBM ([CMKAB18]). PetIBM is a traditional solver using staggered-grid finite difference methods with MPI parallelization and GPU computing. PetIBM simulations in each case study served as baseline data. For all cases, configurations, post-processing scripts, and required Singularity image definitions can be found at reference [Chu22].

This paper is structured as follows: the second section briefly describes the PINN method and an analogy to traditional CFD methods. The third and fourth sections provide our computational experiments of the Taylor-Green vortex in 2D and a 2D laminar cylinder flow with vortex shedding. Most discussions happen in the corresponding case studies. The last section presents the conclusion and discussions that did not fit into either one of the cases.

2. Solving Navier-Stokes equations with PINN

The incompressible Navier-Stokes equations in vector form are composed of the continuity equation:

$$\nabla \cdot \vec{U} = 0 \tag{1}$$

and momentum equations:

$$\frac{\partial \vec{U}}{\partial t} + (\vec{U}\cdot\nabla)\vec{U} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\vec{U} + \vec{g} \tag{2}$$

where $\rho = \rho(\vec{x},t)$, $\nu = \nu(\vec{x},t)$, and $p = p(\vec{x},t)$ are scalar fields denoting density, kinematic viscosity, and pressure, respectively. $\vec{x}$ denotes the spatial coordinate, and $\vec{x} = [x,\ y]^\mathsf{T}$ in two dimensions. The density and viscosity fields are usually known and given, while the pressure field is unknown. $\vec{U} = \vec{U}(\vec{x},t) = [u(x,y,t),\ v(x,y,t)]^\mathsf{T}$ is a vector field for flow velocity. All of them are functions of the spatial coordinate in the computational domain Ω and time before a given limit T. The gravitational field $\vec{g}$ may also be a function of space and time, though it is usually a constant. A solution to the Navier-Stokes equations is subjected to an initial condition and boundary conditions:

$$\begin{cases} \vec{U}(\vec{x},t) = \vec{U}_0(\vec{x}), & \forall \vec{x}\in\Omega, \quad t = 0 \\ \vec{U}(\vec{x},t) = \vec{U}_\Gamma(\vec{x},t), & \forall \vec{x}\in\Gamma, \quad t\in[0,T] \\ p(\vec{x},t) = p_\Gamma(x,t), & \forall \vec{x}\in\Gamma, \quad t\in[0,T] \end{cases} \tag{3}$$

where Γ represents the boundary of the computational domain.

2.1. The PINN method

The basic form of the PINN method ([RPK], [CMW+]) starts from approximating $\vec{U}$ and p with a neural network:

$$\begin{bmatrix}\vec{U}\\ p\end{bmatrix}(\vec{x},t) \approx G(\vec{x},t;\Theta) \tag{4}$$

Here we use a single network that predicts both pressure and velocity fields. It is also possible to use different networks for them separately. Later in this work, we will use G^U and G^p to denote the predicted velocity and pressure from the neural network. Θ at this point represents the free parameters of the network.

To determine the free parameters, Θ, ideally, we hope the approximate solution gives zero residuals for equations (1), (2), and (3). That is

$$\begin{aligned} r_1(\vec{x},t;\Theta) &\equiv \nabla\cdot G^U = 0 \\ r_2(\vec{x},t;\Theta) &\equiv \frac{\partial G^U}{\partial t} + (G^U\cdot\nabla)G^U + \frac{1}{\rho}\nabla G^p - \nu\nabla^2 G^U - \vec{g} = 0 \\ r_3(\vec{x};\Theta) &\equiv G^U_{t=0} - \vec{U}_0 = 0 \\ r_4(\vec{x},t;\Theta) &\equiv G^U - \vec{U}_\Gamma = 0,\ \forall\vec{x}\in\Gamma \\ r_5(\vec{x},t;\Theta) &\equiv G^p - p_\Gamma = 0,\ \forall\vec{x}\in\Gamma \end{aligned} \tag{5}$$

And the set of desired parameter, $\Theta = \theta$, is the common zero root of all the residuals.

The derivatives of G with respect to $\vec{x}$ and t are usually obtained using automatic differentiation. Nevertheless, it is possible to use analytical derivatives when the chosen network architecture is simple enough, as reported by early-day literature ([LLF], [LLQH]).

If residuals in (5) are not complicated, and if the number of the parameters, N_Θ, is small enough, we may numerically find the zero root by solving a system of N_Θ nonlinear equations generated from a suitable set of N_Θ spatial-temporal points. However, the scenario rarely happens as G is usually highly complicated and N_Θ is large. Moreover, we do not even know if such a zero root exists for the equations in (5).

Instead, in PINN, the condition is relaxed. We do not seek the zero root of (5) but just hope to find a set of parameters that make

the residuals sufficiently close to zero. Consider the sum of the l_2 norms of residuals:

$$r(\vec{x},t;\Theta=\theta) \equiv \sum_{i=1}^{5} \|r_i(\vec{x},t;\Theta=\theta)\|^2, \ \forall \begin{cases} x \in \Omega \\ t \in [0,T] \end{cases} \tag{6}$$

The θ that makes residuals closest to zero (or even equal to zero if such θ exists) also makes (6) minimal because $r(\vec{x},t;\Theta) \geq 0$. In other words,

$$\theta = \arg\min_{\Theta} r(\vec{x},t;\Theta) \ \forall \begin{cases} x \in \Omega \\ t \in [0,T] \end{cases} \tag{7}$$

This poses a fundamental difference between the PINN method and traditional CFD schemes, making it potentially more difficult for the PINN method to achieve the same accuracy as the traditional schemes. We will discuss this more in section 3. Note that in practice, each loss term on the right-hand-side of equation (6) is weighted. We ignore the weights here for demonstrating purpose.

To solve (7), theoretically, we can use any number of spatial-temporal points, which eases the need of computational resources, compared to finding the zero root directly. Gradient-descent-based optimizers further reduce the computational cost, especially in terms of memory usage and the difficulty of parallelization. Alternatively, Quasi-Newton methods may work but only when N_Θ is small enough.

However, even though equation (7) may be solvable, it is still a significantly expensive task. While typical data-driven learning requires one back-propagation pass on the derivatives of the loss function, here automatic differentiation is needed to evaluate the derivatives of G with respect to $\vec{x}$ and t. The first-order derivatives require one back-propagation on the network, while the second-order derivatives present in the diffusion term $\nabla^2 G^U$ require an additional back-propagation on the first-order derivatives' computational graph. Finally, to update parameters in an optimizer, the gradients of G with respect to parameters Θ requires another back-propagation on the graph of the second-order derivatives. This all leads to a very large computational graph. We will see the performance of the PINN method in the case studies.

In summary, when viewing the PINN method as supervised machine learning, the inputs of a network are spatial-temporal coordinates, and the outputs are the physical quantities of our interest. The loss or objective functions in PINN are governing equations that regulate how the target physical quantities should behave. The use of governing equations eliminates the need for true answers. A trivial example is using Bernoulli's equation as the loss function, i.e., $loss = \frac{u^2}{2g} + \frac{p}{\rho g} - H_0 + z(x)$, and a neural network predicts the flow speed u and pressure p at a given location x along a streamline. (The gravitational acceleration g, density ρ, energy head H_0, and elevation $z(x)$ are usually known and given.) Such a loss function regulates the relationship between predicted u and p and does not need true answers for the two quantities. Unlike Bernoulli's equation, most governing equations in physics are usually differential equations (e.g., heat equations). The main difference is that now the PINN method needs automatic differentiation to evaluate the loss. Regardless of the forms of governing equations, spatial-temporal coordinates are the only data required during training. Hence, throughout this paper, training data means spatial-temporal points and does not involve any true answers to predicted quantities. (Note in some literature, the PINN method is applied to applications that do need true answers, see [CMW+]. These applications are out of scope here.)

2.2. An analogy to conventional numerical methods

For readers with a background in numerical methods for partial differential equations, we would like to make an analogy between traditional numerical methods and PINN.

In obtaining strong solutions to differential equations, we can describe the solution workflows of most numerical methods with five stages:

1) *Designing the approximate solution with undetermined parameters*
2) *Choosing proper approximation for derivatives*
3) *Obtaining the so-called modified equation by substituting approximate derivatives into the differential equations and initial/boundary conditions*
4) *Generating a system of linear/nonlinear algebraic equations*
5) *Solving the system of equations*

For example, to solve $\nabla U^2(x) = s(x)$, the most naive spectral method ([Tre]) approximates the solution with $U(x) \approx G(x) = \sum_{i=1}^{N} c_i \phi_i(x)$, where c_i represents undetermined parameters, and $\phi_i(x)$ denotes a set of either polynomials, trigonometric functions, or complex exponentials. Next, obtaining the first derivative of U is straightforward—we can just assume $U'(x) \approx G'(x) = \sum_{i=1}^{N} c_i \phi_i'(x)$. The second-order derivative may be more tricky. One can assume $U''(x) \approx G'' = \sum_{i=1}^{N} c_i \phi_i''(x)$. Or, another choice for nodal bases (i.e., when $\phi_i(x)$ is chosen to make $c_i \equiv G(x_i)$) is $U''(x) \approx \sum_{i=1}^{N} c_i G'(x_i)$. Because $\phi_i(x)$ is known, the derivatives are analytical. After substituting the approximate solution and derivatives in to the target differential equation, we need to solve for parameters $c_1, \cdots, c_N$. We do so by selecting N points from the computational domain and creating a system of N linear equations:

$$\begin{bmatrix} \phi_1''(x_1) & \cdots & \phi_N''(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1''(x_N) & \cdots & \phi_N''(x_N) \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_N \end{bmatrix} - \begin{bmatrix} s(x_1) \\ \vdots \\ s(x_N) \end{bmatrix} = 0 \tag{8}$$

Finally, we determine the parameters by solving this linear system. Though this example uses a spectral method, the workflow also applies to many other numerical methods, such as finite difference methods, which can be reformatted as a form of spectral method.

With this workflow in mind, it should be easy to see the analogy between PINN and conventional numerical methods. Aside from using much more complicated approximate solutions, the major difference lies in how to determine the unknown parameters in the approximate solutions. While traditional methods solve the zero-residual conditions, PINN relies on searching the minimal residuals. A secondary difference is how to approximate derivatives. Conventional numerical methods use analytical or numerical differentiation of the approximate solutions, and the PINN methods usually depends on automatic differentiation. This difference may be minor as we are still able to use analytical differentiation for simple network architectures with PINN. However, automatic differentiation is a major factor affecting PINN's performance.

3. Case 1: Taylor-Green vortex: accuracy and performance

3.1. 2D Taylor-Green vortex

The Taylor-Green vortex represents a family of flows with a specific form of analytical initial flow conditions in both 2D

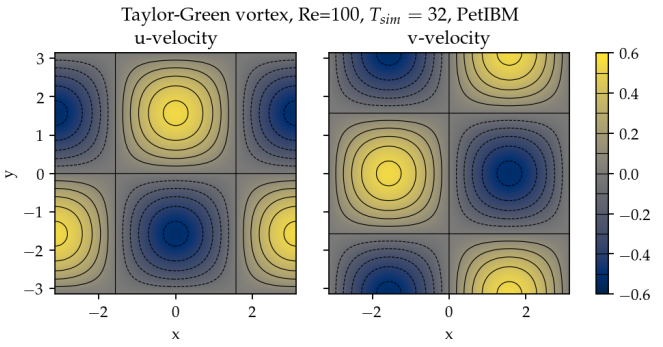


Fig. 1: Contours of u and v at $t = 32$ to demonstrate the solution of 2D Taylor-Green vortex.

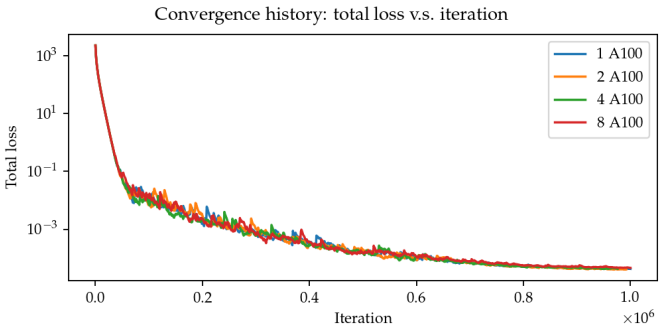


Fig. 2: Total residuals (loss) with respect to training iterations.

and 3D. The 2D Taylor-Green vortex has closed-form analytical solutions with periodic boundary conditions, and hence they are standard benchmark cases for verifying CFD solvers. In this work, we used the following 2D Taylor-Green vortex:

$$\begin{cases} u(x,y,t) = V_0 \cos(\frac{x}{L}) \sin(\frac{y}{L}) \exp(-2\frac{\nu}{L^2}t) \\ v(x,y,t) = -V_0 \sin(\frac{x}{L}) \cos(\frac{y}{L}) \exp(-2\frac{\nu}{L^2}t) \\ p(x,y,t) = -\frac{\rho}{4}V_0^2 \left(cos(\frac{2x}{L}) + cos(\frac{2y}{L}) \right) \exp(-4\frac{\nu}{L^2}t) \end{cases} \tag{9}$$

where V_0 represents the peak (and also the lowest) velocity at $t = 0$. Other symbols carry the same meaning as those in section 2.

The periodic boundary conditions were applied to $x = -L\pi$, $x = L\pi$, $y = -L\pi$, and $y = L\pi$. We used the following parameters in this work: $V_0 = L = \rho = 1.0$ and $\nu = 0.01$. These parameters correspond to Reynolds number $Re = 100$. Figure 1 shows a snapshot of velocity at $t = 32$.

3.2. Solver and runtime configurations

The neural network used in the PINN solver is a fully-connected neural network with 6 hidden layers and 256 neurons per layer. The activation functions are SiLU ([HG]). We used Adam for optimization, and its initial parameters are the defaults from PyTorch. The learning rate exponentially decayed through PyTorch's `ExponentialLR` with `gamma` equal to $0.95^{1/10000}$. Note we did not conduct hyperparameter optimization, given the computational cost. The hyperparameters are mostly the defaults used by the 3D Taylor-Green example in Modulus ([noa]).

The training data were simply spatial-temporal coordinates. Before the training, the PINN solver pre-generated 18,432,000 spatial-temporal points to evaluate the residuals of the Navier-Stokes equations (the r_1 and r_2 in equation (5)). These training points were randomly chosen from the spatial domain $[-\pi,\pi] \times [-\pi,\pi]$ and temporal domain $(0,100]$. The solver used only 18,432 points in each training iteration, making it a batch training. For the residual of the initial condition (the r_3), the solver also pre-generated 18,432,000 random spatial points and used only 18,432 per iteration. Note that for r_3, the points were distributed in space only because $t = 0$ is a fixed condition. Because of the periodic boundary conditions, the solver did not require any training points for r_4 and r_5.

The hardware used for the PINN solver was a single node of NVIDIA's DGX-A100. It was equipped with 8 A100 GPUs (80GB variants). We carried out the training using different numbers of GPUs to investigate the performance of the PINN solver. All cases were trained up to 1 million iterations. Note that the parallelization was done with weak scaling, meaning increasing the number of GPUs would not reduce the workload of each GPU. Instead, increasing the number of GPUs would increase the total and per-iteration numbers of training points. Therefore, our expected outcome was that all cases required about the same wall time to finish, while the residual from using 8 GPUs would converge the fastest.

After training, the PINN solver's prediction errors (i.e., accuracy) were evaluated on cell centers of a 512×512 Cartesian mesh against the analytical solution. With these spatially distributed errors, we calculated the L_2 error norm for a given t:

$$L_2 = \sqrt{\int_\Omega error(x,y)^2 \mathrm{d}\Omega} \approx \sqrt{\sum_i \sum_j error_{i,j}^2 \Delta\Omega_{i,j}} \tag{10}$$

where i and j here are the indices of a cell center in the Cartesian mesh. $\Delta\Omega_{i,j}$ is the corresponding cell area, $4\pi^2/512^2$ in this case.

We compared accuracy and performance against results using PetIBM. All PetIBM simulations in this section were done with 1 K40 GPU and 6 CPU cores (Intel i7-5930K) on our old lab workstation. We carried out 7 PetIBM simulations with different spatial resolutions: $2^k \times 2^k$ for $k = 4,5,\ldots,10$. The time step size for each spatial resolution was $\Delta t = 0.1/2^{k-4}$.

A special note should be made here: the PINN solver used single-precision floats, while PetIBM used double-precision floats. It might sound unfair. However, this discrepancy does not change the qualitative findings and conclusions, as we will see later.

3.3. Results

Figure 2 shows the convergence history of the total residuals (equation (6)). Using more GPUs in weak scaling (i.e., more training points) did not accelerate the convergence, contrary to what we expected. All cases converged at a similar rate. Though without a quantitative criterion or justification, we considered that further training would not improve the accuracy. Figure 3 gives a visual taste of what the predictions from the neural network look like.

The result visually agrees with that in figure 1. However, as shown in figure 4, the error magnitudes from the PINN solver are much higher than those from PetIBM. Figure 4 shows the prediction errors with respect to t. We only present the error on the u velocity as those for v and p are similar. The accuracy of the PINN solver is similar to that of the 16×16 simulation with

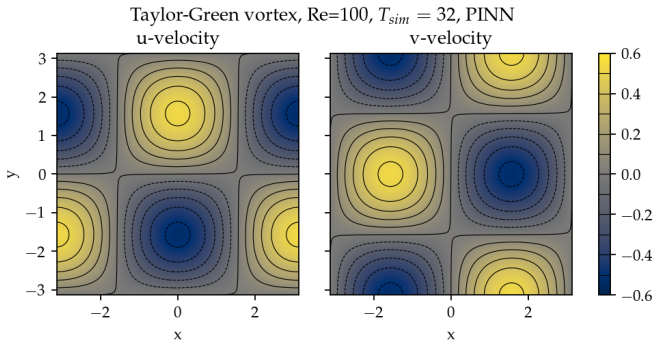


Fig. 3: *Contours of u and v at $t = 32$ from the PINN solver.*

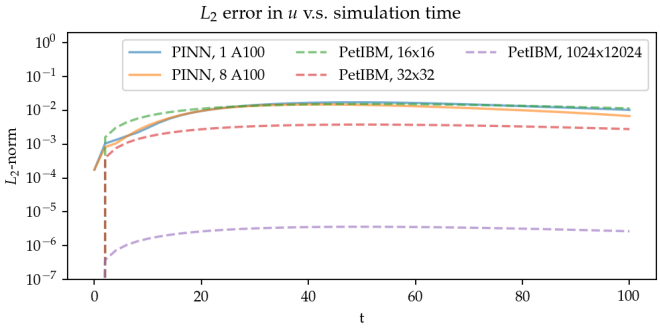


Fig. 4: *L_2 error norm versus simulation time.*

PetIBM. Using more GPUs, which implies more training points, does not improve the accuracy.

Regardless of the magnitudes, the trends of the errors with respect to t are similar for both PINN and PetIBM. For PetIBM, the trend shown in figure 4 indicates that the temporal error is bounded, and the scheme is stable. However, this concept does not apply to PINN as it does not use any time-marching schemes. What this means for PINN is still unclear to us. Nevertheless, it shows that PINN is able to propagate the influence of initial conditions to later times, which is a crucial factor for solving hyperbolic partial differential equations.

Figure 5 shows the computational cost of PINN and PetIBM in terms of the desired accuracy versus the required wall time. We only show the PINN results of 8 A100 GPUs on this figure. We believe this type of plot may help evaluate the computational cost in engineering applications. According to the figure, for example, achieving an accuracy of 10^{-3} at $t = 2$ requires less than 1 second for PetIBM with 1 K40 and 6 CPU cores, but it requires more than 8 hours for PINN with at least 1 A100 GPU.

Table 1 lists the wall time per 1 thousand iterations and the scaling efficiency. As indicated previously, weak scaling was used in PINN, which follows most machine learning applications.

	1 GPUs	2 GPUs	4 GPUs	8 GPUs
Time (sec/1k iters)	85.0	87.7	89.1	90.1
Efficiency (%)	100	97	95	94

TABLE 1: *Weak scaling performance of the PINN solver using NVIDIA A100-80GB GPUs*

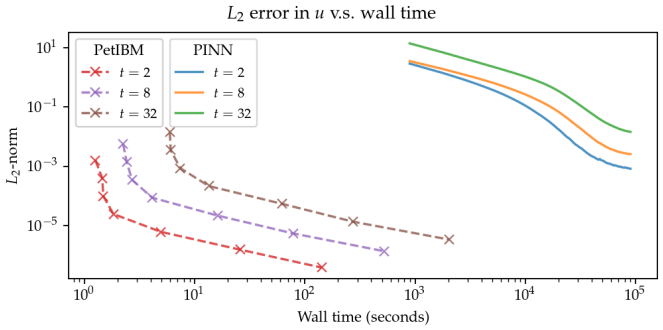


Fig. 5: *L_2 error norm versus wall time.*

3.4. Discussion

A notice should be made regarding the results: we do not claim that these results represent the most optimized configuration of the PINN method. Neither do we claim the qualitative conclusions apply to all other hyperparameter configurations. These results merely reflect the outcomes of our computational experiments with respect to the specific configuration abovementioned. They should be deemed experimental data rather than a thorough analysis of the method's characteristics.

The Taylor-Green vortex serves as a good benchmark case because it reduces the number of required residual constraints: residuals r_4 and r_5 are excluded from r in equation 6. This means the optimizer can concentrate only on the residuals of initial conditions and the Navier-Stokes equations.

Using more GPUs (thus using more training points, i.e., spatio-temporal points) did not speed up the convergence, which may indicate that the per-iteration number of points on a single GPU is already big enough. The number of training points mainly affects the mean gradients of the residual with respect to model parameters, which then will be used to update parameters by gradient-descent-based optimizers. If the number of points is already big enough on a single GPU, then using more points or more GPUs is unlikely to change the mean gradients significantly, causing the convergence solely to rely on learning rates.

The accuracy of the PINN solver was acceptable but not satisfying, especially when considering how much time it took to achieve such accuracy. The low accuracy to some degree was not surprising. Recall the theory in section 2. The PINN method only seeks the minimal residual on the total residual's hyperplane. It does not try to find the zero root of the hyperplane and does not even care whether such a zero root exists. Furthermore, by using a gradient-descent-based optimizer, the resulting minimum is likely just a local minimum. It makes sense that it is hard for the residual to be close to zero, meaning it is hard to make errors small.

Regarding the performance result in figure 5, we would like to avoid interpreting the result as one solver being better than the other one. The proper conclusion drawn from the figure should be as follows: when using the PINN solver as a CFD simulator for a specific flow condition, PetIBM outperforms the PINN solver. As stated in section 1, the PINN method can solve flows under different flow parameters in one run—a capability that PetIBM does not have. The performance result in figure 5 only considers a limited application of the PINN solver.

One issue for this case study was how to fairly compare the PINN solver and PetIBM, especially when investigating the accuracy versus the workload/problem size or time-to-solution

versus problem size. Defining the problem size in PINN is not as straightforward as we thought. Let us start with degrees of freedom—in PINN, it is called the number of model parameters, and in traditional CFD solvers, it is called the number of unknowns. The PINN solver and traditional CFD solvers are all trying to determine the free parameters in models (that is, approximate solutions). Hence, the number of degrees of freedom determines the problem sizes and workloads. However, in PINN, problem sizes and workloads do not depend on degrees of freedom solely. The number of training points also plays a critical role in workloads. We were not sure if it made sense to define a problem size as the sum of the per-iteration number of training points and the number of model parameters. For example, 100 model parameters plus 100 training points is not equivalent to 150 model parameters plus 50 training points in terms of workloads. So without a proper definition of problem size and workload, it was not clear how to fairly compare PINN and traditional CFD methods.

Nevertheless, the gap between the performances of PINN and PetIBM is too large, and no one can argue that using other metrics would change the conclusion. Not to mention that the PINN solver ran on A100 GPUs, while PetIBM ran on a single K40 GPU in our lab, a product from 2013. This is also not a surprising conclusion because, as indicated in section 2, the use of automatic differentiation for temporal and spatial derivatives results in a huge computational graph. In addition, the PINN solver uses gradient-descent based method, which is a first-order method and limits the performance.

Weak scaling is a natural choice of the PINN solver when it comes to distributed computing. As we don't know a proper way to define workload, simply copying all model parameters to all processes and using the same number of training points on all processes works well.

4. Case 2: 2D cylinder flows: harder than we thought

This case study shows what really made us frustrated: a 2D cylinder flow at Reynolds number $Re = 200$. We failed to even produce a solution that qualitatively captures the key physical phenomenon of this flow: vortex shedding.

4.1. Problem description

The computational domain is $[-8,25] \times [-8,8]$, and a cylinder with a radius of 0.5 sits at coordinate $(0,0)$. The velocity boundary conditions are $(u,v) = (1,0)$ along $x = -8$, $y = -8$, and $y = 8$. On the cylinder surface is the no-slip condition, i.e., $(u,v) = (0,0)$. At the outlet ($x = 25$), we enforced a pressure boundary condition $p = 0$. The initial condition is $(u,v) = (0,0)$. Note that this initial condition is different from most traditional CFD simulations. Conventionally, CFD simulations use $(u,v) = (1,0)$ for cylinder flows. A uniform initial condition of $u = 1$ does not satisfy the Navier-Stokes equations due to the no-slip boundary on the cylinder surface. Conventional CFD solvers are usually able to correct the solution during time-marching by propagating boundary effects into the domain through numerical schemes' stencils. In our experience, using $u = 1$ or $u = 0$ did not matter for PINN because both did not give reasonable results. Nevertheless, the PINN solver's results shown in this section were obtained using a uniform $u = 0$ for the initial condition.

The density, ρ, is one, and the kinematic viscosity is $\nu = 0.005$. These parameters correspond to Reynolds number $Re = 200$. Figure 6 shows the velocity and vorticity snapshots at $t = 200$. As shown in the figure, this type of flow displays a phenomenon called vortex shedding. Though vortex shedding makes the flow always unsteady, after a certain time, the flow reaches a periodic stage and the flow pattern repeats after a certain period.

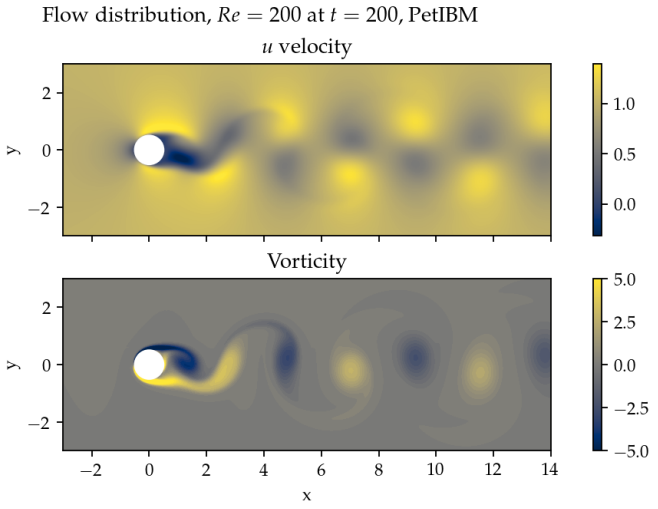


Fig. 6: *Demonstration of velocity and vorticity fields at $t = 200$ from a PetIBM simulation.*

The Navier-Stokes equations can be deemed as a dynamical system. Instability appears in the flow under some flow conditions and responds to small perturbations, causing the vortex shedding. In nature, the vortex shedding comes from the uncertainty and perturbation existing everywhere. In CFD simulations, the vortex shedding is caused by small numerical and rounding errors in calculations. Interested readers should consult reference [Wil].

4.2. Solver and runtime configurations

For the PINN solver, we tested with two networks. Both were fully-connected neural networks: one with 256 neurons per layer, while the other one with 512 neurons per layer. All other network configurations were the same as those in section 3, except we allowed human intervention to manually adjust the learning rates during training. Our intention for this case study was to successfully obtain physical solutions from the PINN solver, rather than conducting a performance and accuracy benchmark. Therefore, we would adjust the learning rate to accelerate the convergence or to escape from local minimums. This decision was in line with common machine learning practice. We did not carry out hyperparameter optimization. These parameters were chosen because they work in Modulus' examples and in the Taylor-Green vortex experiment.

The PINN solver pre-generated $40,960,000$ spatial-temporal points from a spatial domain in $[-8,25] \times [-8,8]$ and temporal domain $(0,200]$ to evaluate residuals of the Navier-Stokes equations, and used $40,960$ points per iteration. The number of pre-generated points for the initial condition was $2,048,000$, and the per-iteration number is $2,048$. On each boundary, the numbers of pre-generated and per-iteration points are 8,192,000 and 8,192. Both cases used 8 A100 GPUs, which scaled these numbers up with a factor of 8. For example, during each iteration, a total of $327,680$ points were actually used to evaluate the Navier-Stokes equations' residuals. Both cases ran up to 64 hours in wall time.

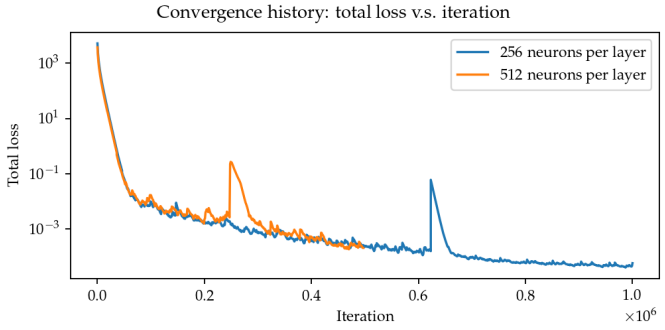


Fig. 7: *Training history of the 2D cylinder flow at $Re = 200$.*

One PetIBM simulation was carried out as a baseline. This simulation had a spatial resolution of 1485×720, and the time step size is 0.005. Figure 6 was rendered using this simulation. The hardware used was 1 K40 GPU plus 6 cores of i7-5930K CPU. It took about 1.7 hours to finish.

The quantity of interest is the drag coefficient. We consider both the friction drag and pressure drag in the coefficient calculation as follows:

$$C_D = \frac{2}{\rho U_0^2 D} \int_S \left(\rho \nu \frac{\partial \left(\vec{U} \cdot \vec{t} \right)}{\partial \vec{n}} n_y - p n_x \right) \mathrm{d}S \qquad (11)$$

Here, $U_0 = 1$ is the inlet velocity. $\vec{n} = [n_x, n_y]^\mathsf{T}$ and $\vec{t} = [n_y, -n_x]^\mathsf{T}$ are the normal and tangent vectors, respectively. S represents the cylinder surface. The theoretical lift coefficient (C_L) for this flow is zero due to the symmetrical geometry.

4.3. Results

Note, as stated in section 3.4, we deem the results as experimental data under a specific experiment configuration. Hence, we do not claim that the results and qualitative conclusions will apply to other hyperparameter configuration.

Figure 7 shows the convergence history. The bumps in the history correspond to our manual adjustment of the learning rates. After 64 hours of training, the total loss had not converged to an obvious steady value. However, we decided not to continue the training because, as later results will show, it is our judgment call that the results would not be correct even if the training converged.

Figure 8 provides a visualization of the predicted velocity and vorticity at $t = 200$. And in figure 9 are the drag and lift coefficients versus simulation time. From both figures, we couldn't see any sign of vortex shedding with the PINN solver.

We provide a comparison against the values reported by others in table 2. References [GS74] and [For80] calculate the drag coefficients using steady flow simulations, which were popular decades ago because of their inexpensive computational costs. The actual flow is not a steady flow, and these steady-flow coefficient values are lower than unsteady-flow predictions. The drag coefficient from the PINN solver is closer to the steady-flow predictions.

4.4. Discussion

While researchers may be interested in why the PINN solver behaves like a steady flow solver, in this section, we would like to focus more on the user experience and the usability of PINN in

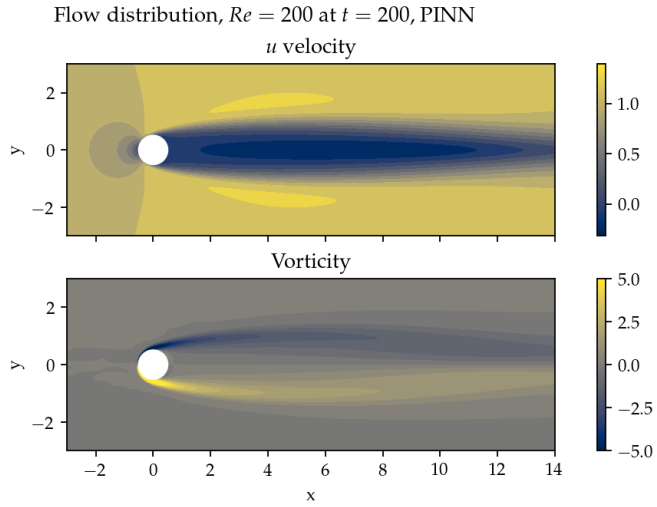


Fig. 8: *Velocity and vorticity at $t = 200$ from PINN.*

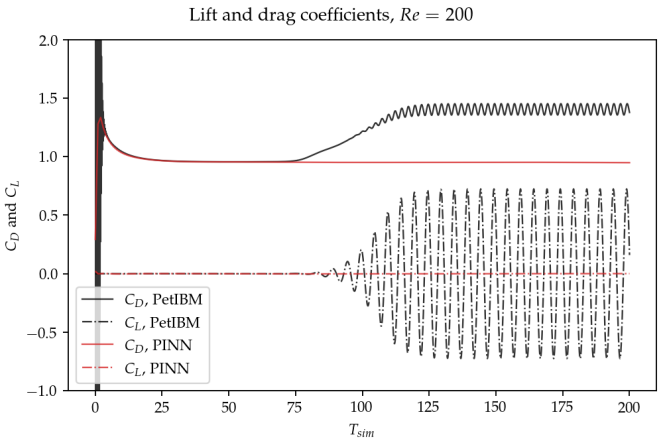


Fig. 9: *Drag and lift coefficients with respect to t*

practice. Our viewpoints may be subjective, and hence we leave them here in the discussion.

Allow us to start this discussion with a hypothetical situation. If one asks why we chose such a spatial and temporal resolution for a conventional CFD simulation, we have mathematical or physical reasons to back our decision. However, if the person asks why we chose 6 hidden layers and 256 neurons per layer, we will not be able to justify it. "It worked in another case!" is probably the best answer we can offer. The situation also indicates that we have systematic approaches to improve a conventional simulation but can only improve PINN's results through computer experiments.

Most traditional numerical methods have rigorous analytical derivations and analyses. Each parameter used in a scheme has a meaning or a purpose in physical or numerical aspects. The simplest example is the spatial resolution in the finite difference method, which controls the truncation errors in derivatives. Or,

		Unsteady simulations		Steady simulations	
PetIBM	PINN	[DSY07]	[RKM09]	[GS74]	[For80]
1.38	0.95	1.25	1.34	0.97	0.83

TABLE 2: *Comparison of drag coefficients, C_D*

the choice of the limiters in finite volume methods, used to inhibit the oscillation in solutions. So when a conventional CFD solver produces unsatisfying or even non-physical results, practitioners usually have systematic approaches to identify the cause or improve the outcomes. Moreover, when necessary, practitioners know how to balance the computational cost and the accuracy, which is a critical point for using computer-aided engineering. Engineering always concerns the costs and outcomes.

On the other hand, the PINN method lacks well-defined procedures to control the outcome. For example, we know the numbers of neurons and layers control the degrees of freedom in a model. With more degrees of freedom, a neural network model can approximate a more complicated phenomenon. However, when we feel that a neural network is not complicated enough to capture a physical phenomenon, what strategy should we use to adjust the neurons and layers? Should we increase neurons or layers first? By how much?

Moreover, when it comes to something non-numeric, it is even more challenging to know what to use and why to use it. For instance, what activation function should we use and why? Should we use the same activation everywhere? Not to mention that we are not yet even considering a different network architecture here.

Ultimately, are we even sure that increasing the network's complexity is the right path? Our assumption that the network is not complicated enough may just be wrong.

The following situation happened in this case study. Before we realized the PINN solver behaved like a steady-flow solver, we attributed the cause to model complexity. We faced the problem of how to increase the model complexity systematically. Theoretically, we could follow the practice of the design of experiments (e.g., through grid search or Taguchi methods). However, given the computational cost and the number of hyperparameters/options of PINN, a proper design of experiments is not affordable for us. Furthermore, the design of experiments requires the outcome to change with changes in inputs. In our case, the vortex shedding remains absent regardless of how we changed hyperparameters.

Let us move back to the flow problem to conclude this case study. The model complexity may not be the culprit here. Vortex shedding is the product of the dynamical systems of the Navier-Stokes equations and the perturbations from numerical calculations (which implicitly mimic the perturbations in nature). Suppose the PINN solver's prediction was the steady-state solution to the flow. We may need to introduce uncertainties and perturbations in the neural network or the training data, such as a perturbed initial condition described in [LD15]. As for why PINN predicts the steady-state solution, we cannot answer it currently.

5. Further discussion and conclusion

Because of the widely available deep learning libraries, such as PyTorch, and the ease of Python, implementing a PINN solver is relatively more straightforward nowadays. This may be one reason why the PINN method suddenly became so popular in recent years. This paper does not intend to discourage people from trying the PINN method. Instead, we share our failures and frustration using PINN so that interested readers may know what immediate challenges should be resolved for PINN.

Our paper is limited to using the PINN solver as a replacement for traditional CFD solvers. However, as the first section indicates, PINN can do more than solving one specific flow under specific flow parameters. Moreover, PINN can also work with traditional CFD solvers. The literature shows researchers have shifted their attention to hybrid-mode applications. For example, in [JEA+20], the authors combined the concept of PINN and a traditional CFD solver to train a model that takes in low-resolution CFD simulation results and outputs high-resolution flow fields.

For people with a strong background in numerical methods or CFD, we would suggest trying to think out of the box. During our work, we realized our mindset and ideas were limited by what we were used to in CFD. An example is the initial conditions. We were used to only having one set of initial conditions when the temporal derivative in differential equations is only first-order. However, in PINN, nothing limits us from using more than one initial condition. We can generate results at $t = 0, 1, \ldots, t_n$ using a traditional CFD solver and add the residuals corresponding to these time snapshots to the total residual, so the PINN method may perform better in predicting $t > t_n$. In other words, the PINN solver becomes the traditional CFD solvers' replacement only for $t > t_n$ ([noa]).

As discussed in [THM+], solving partial differential equations with deep learning is still a work-in-progress. It may not work in many situations. Nevertheless, it does not mean we should stay away from PINN and discard this idea. Stepping away from a new thing gives zero chance for it to evolve, and we will never know if PINN can be improved to a mature state that works well. Of course, overly promoting its bright side with only success stories does not help, either. Rather, we should honestly face all troubles, difficulties, and challenges. Knowing the problem is the first step to solving it.

Acknowledgements

We appreciate the support by NVIDIA, through sponsoring the access to its high-performance computing cluster.

References

[Chu22] Pi-Yueh Chuang. barbagroup/scipy-2022-repro-pack: 20220530, May 2022. URL: https://doi.org/10.5281/zenodo.6592457, doi:10.5281/zenodo.6592457.

[CMKAB18] Pi-Yueh Chuang, Olivier Mesnard, Anush Krishnan, and Lorena A. Barba. PetIBM: toolbox and applications of the immersed-boundary method on distributed-memory architectures. *Journal of Open Source Software*, 3(25):558, May 2018. URL: http://joss.theoj.org/papers/10.21105/joss.00558, doi:10.21105/joss.00558.

[CMW+] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: a review. 37(12):1727–1738. URL: https://link.springer.com/10.1007/s10409-021-01148-1, doi:10.1007/s10409-021-01148-1.

[DPT] M. W. M. G. Dissanayake and N. Phan-Thien. Neural-network-based approximations for solving partial differential equations. 10(3):195–201. URL: https://onlinelibrary.wiley.com/doi/10.1002/cnm.1640100303, doi:10.1002/cnm.1640100303.

[DSY07] Jian Deng, Xue-Ming Shao, and Zhao-Sheng Yu. Hydrodynamic studies on two traveling wavy foils in tandem arrangement. *Physics of Fluids*, 19(11):113104, November 2007. URL: http://aip.scitation.org/doi/10.1063/1.2814259, doi:10.1063/1.2814259.

[DZ] Yifan Du and Tamer A. Zaki. Evolutional deep neural network. 104(4):045303. URL: https://link.aps.org/doi/10.1103/PhysRevE.104.045303, doi:10.1103/PhysRevE.104.045303.

[For80] Bengt Fornberg. A numerical study of steady viscous flow past a circular cylinder. *Journal of Fluid Mechanics*, 98(04):819, June 1980. URL: http://www.journals.cambridge.org/abstract_S0022112080000419, doi:10.1017/S0022112080000419.

[FS] William E. Faller and Scott J. Schreck. Unsteady fluid mechanics applications of neural networks. 34(1):48–55. URL: http://arc.aiaa.org/doi/10.2514/2.2134, doi:10.2514/2.2134.

[GS74] V.A. Gushchin and V.V. Shchennikov. A numerical method of solving the navier-stokes equations. *USSR Computational Mathematics and Mathematical Physics*, 14(2):242–250, January 1974. URL: https://linkinghub.elsevier.com/retrieve/pii/0041555374900615, doi:10.1016/0041-5553(74)90061-5.

[Hao] Karen Hao. AI has cracked a key mathematical puzzle for understanding our world. URL: https://www.technologyreview.com/2020/10/30/1011435/ai-fourier-neural-network-cracks-navier-stokes-and-partial-differential-equations/.

[HG] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). Publisher: arXiv Version Number: 4. URL: https://arxiv.org/abs/1606.08415, doi:10.48550/ARXIV.1606.08415.

[Hor] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. 4(2):251–257. URL: https://linkinghub.elsevier.com/retrieve/pii/089360809190009T, doi:10.1016/0893-6080(91)90009-T.

[JEA+20] Chiyu "Max" Jiang, Soheil Esmaeilzadeh, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A. Tchelepi, Philip Marcus, Mr Prabhat, and Anima Anandkumar. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. doi:10.1109/SC41405.2020.00013.

[KDYI] Hasan Karali, Umut M. Demirezen, Mahmut A. Yukselen, and Gokhan Inalhan. A novel physics informed deep learning method for simulation-based modelling. In *AIAA Scitech 2021 Forum*. American Institute of Aeronautics and Astronautics. URL: https://arc.aiaa.org/doi/10.2514/6.2021-0177, doi:10.2514/6.2021-0177.

[LD15] Mouna Laroussi and Mohamed Djebbi. Vortex Shedding for Flow Past Circular Cylinder: Effects of Initial Conditions. *Universal Journal of Fluid Mechanics*, 3:19–32, 2015.

[LLF] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. 9(5):987–1000. URL: http://ieeexplore.ieee.org/document/712178/, arXiv:physics/9705023, doi:10.1109/72.712178.

[LLQH] Jianyu Li, Siwei Luo, Yingjian Qi, and Yaping Huang. Numerical solution of elliptic partial differential equation using radial basis function neural networks. 16(5):729–734. URL: https://linkinghub.elsevier.com/retrieve/pii/S0893608003000832, doi:10.1016/S0893-6080(03)00083-2.

[LMMK] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. 63(1):208–228. URL: https://epubs.siam.org/doi/10.1137/19M1274067, doi:10.1137/19M1274067.

[LS] Dennis J. Linse and Robert F. Stengel. Identification of aerodynamic coefficients using computational neural networks. 16(6):1018–1025. Publisher: Springer US, Place: Boston, MA. URL: http://link.springer.com/10.1007/0-306-48610-5_9, doi:10.2514/3.21122.

[noa] Modulus. URL: https://docs.nvidia.com/deeplearning/modulus/index.html.

[RKM09] B.N. Rajani, A. Kandasamy, and Sekhar Majumdar. Numerical simulation of laminar flow past a circular cylinder. *Applied Mathematical Modelling*, 33(3):1228–1247, March 2009. arXiv: DOI: 10.1002/fld.1 Publisher: Elsevier Inc. ISBN: 02712091 10970363. URL: http://dx.doi.org/10.1016/j.apm.2008.01.017, doi:10.1016/j.apm.2008.01.017.

[RPK] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. 378:686–707. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999118307125, doi:10.1016/j.jcp.2018.10.045.

[SS] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. 375:1339–1364. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999118305527, doi:10.1016/j.jcp.2018.08.029.

[THM+] Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. Physics-based deep learning. Number: arXiv:2109.05237. URL: http://arxiv.org/abs/2109.05237, arXiv:2109.05237[physics].

[Tre] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Software, environments, tools. Society for Industrial and Applied Mathematics. URL: http://epubs.siam.org/doi/book/10.1137/1.9780898719598, doi:10.1137/1.9780898719598.

[Wil] C. H. K. Williamson. Vortex dynamics in the cylinder wake. 28(1):477–539. URL: http://www.annualreviews.org/doi/10.1146/annurev.fl.28.010196.002401, doi:10.1146/annurev.fl.28.010196.002401.

[WTP] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. 43(5):A3055–A3081. URL: https://epubs.siam.org/doi/10.1137/20M1318043, doi:10.1137/20M1318043.

[WYP] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why PINNs fail to train: A neural tangent kernel perspective. 449:110768. URL: https://linkinghub.elsevier.com/retrieve/pii/S002199912100663X, doi:10.1016/j.jcp.2021.110768.

atoMEC: An open-source average-atom Python code

Timothy J. Callow[‡§*], Daniel Kotik[‡§], Eli Kraisler[¶], Attila Cangi[‡§]

Abstract—Average-atom models are an important tool in studying matter under extreme conditions, such as those conditions experienced in planetary cores, brown and white dwarfs, and during inertial confinement fusion. In the right context, average-atom models can yield results with similar accuracy to simulations which require orders of magnitude more computing time, and thus can greatly reduce financial and environmental costs. Unfortunately, due to the wide range of possible models and approximations, and the lack of open-source codes, average-atom models can at times appear inaccessible. In this paper, we present our open-source average-atom code, atoMEC. We explain the aims and structure of atoMEC to illuminate the different stages and options in an average-atom calculation, and to facilitate community contributions. We also discuss the use of various open-source Python packages in atoMEC, which have expedited its development.

Index Terms—computational physics, plasma physics, atomic physics, materials science

Introduction

The study of matter under extreme conditions — materials exposed to high temperatures, high pressures, or strong electromagnetic fields — is critical to our understanding of many important scientific and technological processes, such as nuclear fusion and various astrophysical and planetary physics phenomena [GFG+16]. Of particular interest within this broad field is the warm dense matter (WDM) regime, which is typically characterized by temperatures in the range of $10^3 - 10^6$ degrees (Kelvin), and densities ranging from dense gases to highly compressed solids ($\sim 0.01 - 1000$ g cm^{-3}) [BDM+20]. In this regime, it is important to account for the quantum mechanical nature of the electrons (and in some cases, also the nuclei). Therefore conventional methods from plasma physics, which either neglect quantum effects or treat them coarsely, are usually not sufficiently accurate. On the other hand, methods from condensed-matter physics and quantum chemistry, which account fully for quantum interactions, typically target the ground-state only, and become computationally intractable for systems at high temperatures.

Nevertheless, there are methods which can, in principle, be applied to study materials at any given temperature and density whilst formally accounting for quantum interactions. These methods are often denoted as "first-principles" because, formally speaking, they yield the exact properties of the system, under certain well-founded theoretical approximations. Density-functional theory (DFT), initially developed as a ground-state theory [HK64], [KS65] but later extended to non-zero temperatures [Mer65], [PPF+11], is one such theory and has been used extensively to study materials under WDM conditions [GDRT14]. Even though DFT reformulates the Schrödinger equation in a computationally efficient manner [Koh99], the cost of running calculations becomes prohibitively expensive at higher temperatures. Formally, it scales as $\mathcal{O}(N^3\tau^3)$, with N the particle number (which usually also increases with temperature) and τ the temperature [CRNB18]. This poses a serious computational challenge in the WDM regime. Furthermore, although DFT is a formally exact theory, in practice it relies on approximations for the so-called "exchange-correlation" energy, which is, roughly speaking, responsible for simulating all the quantum interactions between electrons. Existing exchange-correlation approximations have not been rigorously tested under WDM conditions. An alternative method used in the WDM community is path-integral Monte–Carlo [DGB18], which yields essentially exact properties; however, it is even more limited by computational cost than DFT, and becomes unfeasibly expensive at lower temperatures due to the fermion sign problem.

It is therefore of great interest to reduce the computational complexity of the aforementioned methods. The use of graphics processing units in DFT calculations is becomingly increasingly common, and has been shown to offer significant speed-ups relative to conventional calculations using central processing units [MED11], [JFC+13]. Some other examples of promising developments to reduce the cost of DFT calculations include machine-learning-based solutions [SRH+12], [BVL+17], [EFP+21] and stochastic DFT [CRNB18], [BNR13]. However, in this paper, we focus on an alternative class of models known as "average-atom" models. Average-atom models have a long history in plasma physics [CHKC22]: they account for quantum effects, typically using DFT, but reduce the complex system of interacting electrons and nuclei to a single atom immersed in a plasma (the "average" atom). An illustration of this principle (reduced to two dimensions for visual purposes) is shown in Fig. 1. This significantly reduces the cost relative to a full DFT simulation, because the particle number is restricted to the number of electrons per nucleus, and spherical symmetry is exploited to reduce the three-dimensional problem to one dimension.

Naturally, to reduce the complexity of the problem as described, various approximations must be introduced. It is important to understand these approximations and their limitations for average-atom models to have genuine predictive capabilities. Unfortunately, this is not always the case: although average-atom

* Corresponding author: t.callow@hzdr.de
‡ Center for Advanced Systems Understanding (CASUS), D-02826 Görlitz, Germany
§ Helmholtz-Zentrum Dresden-Rossendorf, D-01328 Dresden, Germany
¶ Fritz Haber Center for Molecular Dynamics and Institute of Chemistry, The Hebrew University of Jerusalem, 9091401 Jerusalem, Israel

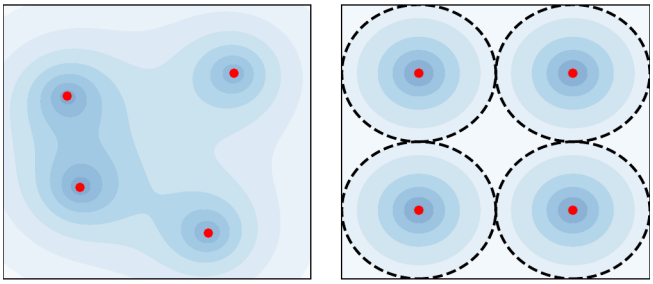


Fig. 1: *Illustration of the average-atom concept. The many-body and fully-interacting system of electron density (shaded blue) and nuclei (red points) on the left is mapped into the much simpler system of independent atoms on the right. Any of these identical atoms represents the "average-atom". The effects of interaction from neighboring atoms are implicitly accounted for in an approximate manner through the choice of boundary conditions.*

models share common concepts, there is no unique formal theory underpinning them. Therefore a variety of models and codes exist, and it is not typically clear which models can be expected to perform most accurately under which conditions. In a previous paper [CHKC22], we addressed this issue by deriving an average-atom model from first principles, and comparing the impact of different approximations within this model on some common properties.

In this paper, we focus on computational aspects of average-atom models for WDM. We introduce atoMEC [CKTS+21]: an open-source average-**ato**m code for studying **M**atter under **E**xtreme **C**onditions. One of the main aims of atoMEC is to improve the accessibility and understanding of average-atom models. To the best of our knowledge, open-source average-atom codes are in scarce supply: with atoMEC, we aim to provide a tool that people can use to run average-atom simulations and also to add their own models, which should facilitate comparisons of different approximations. The relative simplicity of average-atom codes means that they are not only efficient to run, but also efficient to develop: this means, for example, that they can be used as a test-bed for new ideas that could be later implemented in full DFT codes, and are also accessible to those without extensive prior expertise, such as students. atoMEC aims to facilitate development by following good practice in software engineering (for example extensive documentation), a careful design structure, and of course through the choice of Python and its widely used scientific stack, in particular the NumPy [HMvdW+20] and SciPy [VGO+20] libraries.

This paper is structured as follows: in the next section, we briefly review the key theoretical points which are important to understand the functionality of atoMEC, assuming no prior physical knowledge of the reader. Following that, we present the key functionality of atoMEC, discuss the code structure and algorithms, and explain how these relate to the theoretical aspects introduced. Finally, we present an example case study: we consider helium under the conditions often experienced in the outer layers of a white dwarf star, and probe the behavior of a few important properties, namely the band-gap, pressure, and ionization degree.

Theoretical background

Properties of interest in the warm dense matter regime include the equation-of-state data, which is the relation between the density, energy, temperature and pressure of a material [HRD08]; the mean ionization state and the electron ionization energies, which tell us about how tightly bound the electrons are to the nuclei; and the electrical and thermal conductivities. These properties yield information pertinent to our understanding of stellar and planetary physics, the Earth's core, inertial confinement fusion, and more besides. To exactly obtain these properties, one needs (in theory) to determine the thermodynamic ensemble of the quantum states (the so-called *wave-functions*) representing the electrons and nuclei. Fortunately, they can be obtained with reasonable accuracy using models such as average-atom models; in this section, we elaborate on how this is done.

We shall briefly review the key theory underpinning the type of average-atom model implemented in atoMEC. This is intended for readers without a background in quantum mechanics, to give some context to the purposes and mechanisms of the code. For a comprehensive derivation of this average-atom model, we direct readers to Ref. [CHKC22]. The average-atom model we shall describe falls into a class of models known as *ion-sphere* models, which are the simplest (and still most widely used) class of average-atom model. There are alternative (more advanced) classes of model such as *ion-correlation* [Roz91] and *neutral pseudo-atom* models [SS14] which we have not yet implemented in atoMEC, and thus we do not elaborate on them here.

As demonstrated in Fig. 1, the idea of the ion-sphere model is to map a fully-interacting system of many electrons and nuclei into a set of independent atoms which do not interact explicitly with any of the other spheres. Naturally, this depends on several assumptions and approximations, but there is formal justification for such a mapping [CHKC22]. Furthermore, there are many examples in which average-atom models have shown good agreement with more accurate simulations and experimental data [FB19], which further justifies this mapping.

Although the average-atom picture is significantly simplified relative to the full many-body problem, even determining the wave-functions and their ensemble weights for an atom at finite temperature is a complex problem. Fortunately, DFT reduces this complexity further, by establishing that the electron *density* — a far less complex entity than the wave-functions — is sufficient to determine all physical observables. The most popular formulation of DFT, known as Kohn–Sham DFT (KS-DFT) [KS65], allows us to construct the *fully-interacting* density from a *non-interacting* system of electrons, simplifying the problem further still. Due to the spherical symmetry of the atom, the non-interacting electrons — known as KS electrons (or KS orbitals) — can be represented as a wave-function that is a product of radial and angular components,

$$\phi_{nlm}(\mathbf{r}) = X_{nl}(r) Y_l^m(\theta, \phi)\,, \tag{1}$$

where n, l, and m are the *quantum numbers* of the orbitals, which come from the fact that the wave-function is an eigenfunction of the Hamiltonian operator, and $Y_l^m(\theta,\phi)$ are the spherical harmonic functions.[1] The radial coordinate r represents the absolute distance from the nucleus.

1. Please note that the notation in Eq. (1) does not imply Einstein summation notation. All summations in this paper are written explicitly; Einstein summation notation is not used.

We therefore only need to determine the radial KS orbitals $X_{nl}(r)$. These are determined by solving the radial KS equation, which is similar to the Schrödinger equation for a non-interacting system, with an additional term in the potential to mimic the effects of electron-electron interaction (within the single atom). The radial KS equation is given by:

$$\left[-\left(\frac{\mathrm{d}^2}{\mathrm{d}r^2}+\frac{2}{r}\frac{\mathrm{d}}{\mathrm{d}r}-\frac{l(l+1)}{r^2}\right)+v_\mathrm{s}[n](r)\right]X_{nl}(r)=\varepsilon_{nl}X_{nl}(r). \quad (2)$$

We have written the above equation in a way that emphasizes that it is an eigenvalue equation, with the eigenvalues ε_{nl} being the energies of the KS orbitals.

On the left-hand side, the terms in the round brackets come from the kinetic energy operator acting on the orbitals. The $v_\mathrm{s}[n](r)$ term is the KS potential, which itself is composed of three different terms,

$$v_\mathrm{s}[n](r)=-\frac{Z}{r}+4\pi\int_0^{R_\mathrm{WS}}\mathrm{d}x\frac{n(x)x^2}{\max(r,x)}+\frac{\delta F_\mathrm{xc}[n]}{\delta n(r)}, \quad (3)$$

where R_WS is the radius of the atomic sphere, $n(r)$ is the electron density, Z the nuclear charge, and $F_\mathrm{xc}[n]$ the exchange-correlation free energy functional. Thus the three terms in the potential are respectively the electron-nuclear attraction, the classical Hartree repulsion, and the exchange-correlation (xc) potential.

We note that the KS potential and its constituents are functionals of the electron density $n(r)$. Were it not for this dependence on the density, solving Eq. 2 just amounts to solving an ordinary linear differential equation (ODE). However, the electron density is in fact constructed from the orbitals in the following way,

$$n(r)=2\sum_{nl}(2l+1)f_{nl}(\varepsilon_{nl},\mu,\tau)|X_{nl}(r)|^2, \quad (4)$$

where $f_{nl}(\varepsilon_{nl},\mu,\tau)$ is the Fermi–Dirac distribution, given by

$$f_{nl}(\varepsilon_{nl},\mu,\tau)=\frac{1}{1+e^{(\varepsilon_{nl}-\mu)/\tau}}, \quad (5)$$

where τ is the temperature, and μ is the chemical potential, which is determined by fixing the number of electrons to be equal to a pre-determined value N_e (typically equal to the nuclear charge Z). The Fermi–Dirac distribution therefore assigns weights to the KS orbitals in the construction of the density, with the weight depending on their energy.

Therefore, the KS potential that determines the KS orbitals via the ODE (2), is itself dependent on the KS orbitals. Consequently, the KS orbitals and their dependent quantities (the density and KS potential) must be determined via a so-called self-consistent field (SCF) procedure. An initial guess for the orbitals, $X^0_{nl}(r)$, is used to construct the initial density $n^0(r)$ and potential $v^0_\mathrm{s}(r)$. The ODE (2) is then solved to update the orbitals. This process is iterated until some appropriately chosen quantities — in atoMEC the total free energy, density and KS potential — are converged, i.e. $n^{i+1}(r)=n^i(r)$, $v^{i+1}_\mathrm{s}(r)=v^i_\mathrm{s}(r)$, $F^{i+1}=F^i$, within some reasonable numerical tolerance. In Fig. 2, we illustrate the life-cycle of the average-atom model described so far, including the SCF procedure. On the left-hand side of this figure, we show the physical choices and mathematical operations, and on the right-hand side, the representative classes and functions in atoMEC. In the following section, we shall discuss some aspects of this figure in more detail.

Some quantities obtained from the completion of the SCF procedure are directly of interest. For example, the energy eigenvalues ε_{nl} are related to the electron ionization energies, i.e. the amount of energy required to excite an electron bound to the nucleus to being a free (conducting) electron. These predicted ionization energies can be used, for example, to help understand ionization potential depression, an important but somewhat controversial effect in WDM [STJ+14]. Another property that can be straightforwardly obtained from the energy levels and their occupation numbers is the mean ionization state $\bar{Z}$[2],

$$\bar{Z}=\sum_{n,l}(2l+1)f_{nl}(\varepsilon_{nl},\mu,\tau) \quad (6)$$

which is an important input parameter for various models, such as adiabats which are used to model inertial confinement fusion [KDF+11].

Various other interesting properties can also be calculated following some post-processing of the output of an SCF calculation, for example the pressure exerted by the electrons and ions. Furthermore, response properties, i.e. those resulting from an external perturbation like a laser pulse, can also be obtained from the output of an SCF cycle. These properties include, for example, electrical conductivities [Sta16] and dynamical structure factors [SPS+14].

Code structure and details

In the following sections, we describe the structure of the code in relation to the physical problem being modeled. Average-atom models typically rely on various parameters and approximations. In atoMEC, we have tried to structure the code in a way that makes clear which parameters come from the physical problem studied compared to choices of the model and numerical or algorithmic choices.

atoMEC.Atom: Physical parameters

The first step of any simulation in WDM (which also applies to simulations in science more generally) is to define the physical parameters of the problem. These parameters are unique in the sense that, if we had an exact method to simulate the real system, then for each combination of these parameters there would be a unique solution. In other words, regardless of the model — be it average atom or a different technique — these parameters are always required and are independent of the model.

In average-atom models, there are typically three parameters defining the physical problem, which are:

- the **atomic species**;
- the **temperature** of the material, τ;
- the **mass density** of the material, ρ_m.

The mass density also directly corresponds to the mean distance between two nuclei (atomic centers), which in the average-atom model is equal to twice the radius of the atomic sphere, R_WS. An additional physical parameter not mentioned above is the **net charge** of the material being considered, i.e. the difference between the nuclear charge Z and the electron number N_e. However, we usually assume zero net charge in average-atom simulations (i.e. the number of electrons is equal to the atomic charge).

In atoMEC, these physical parameters are controlled by the `Atom` object. As an example, we consider aluminum under ambient conditions, i.e. at room temperature, $\tau = 300$ K, and normal metallic density, $\rho_\mathrm{m} = 2.7$ g cm^{-3}. We set this up as:

2. The summation in Eq. (6) is often shown as an integral because the energies above a certain threshold form a continuous distribution (in most models).

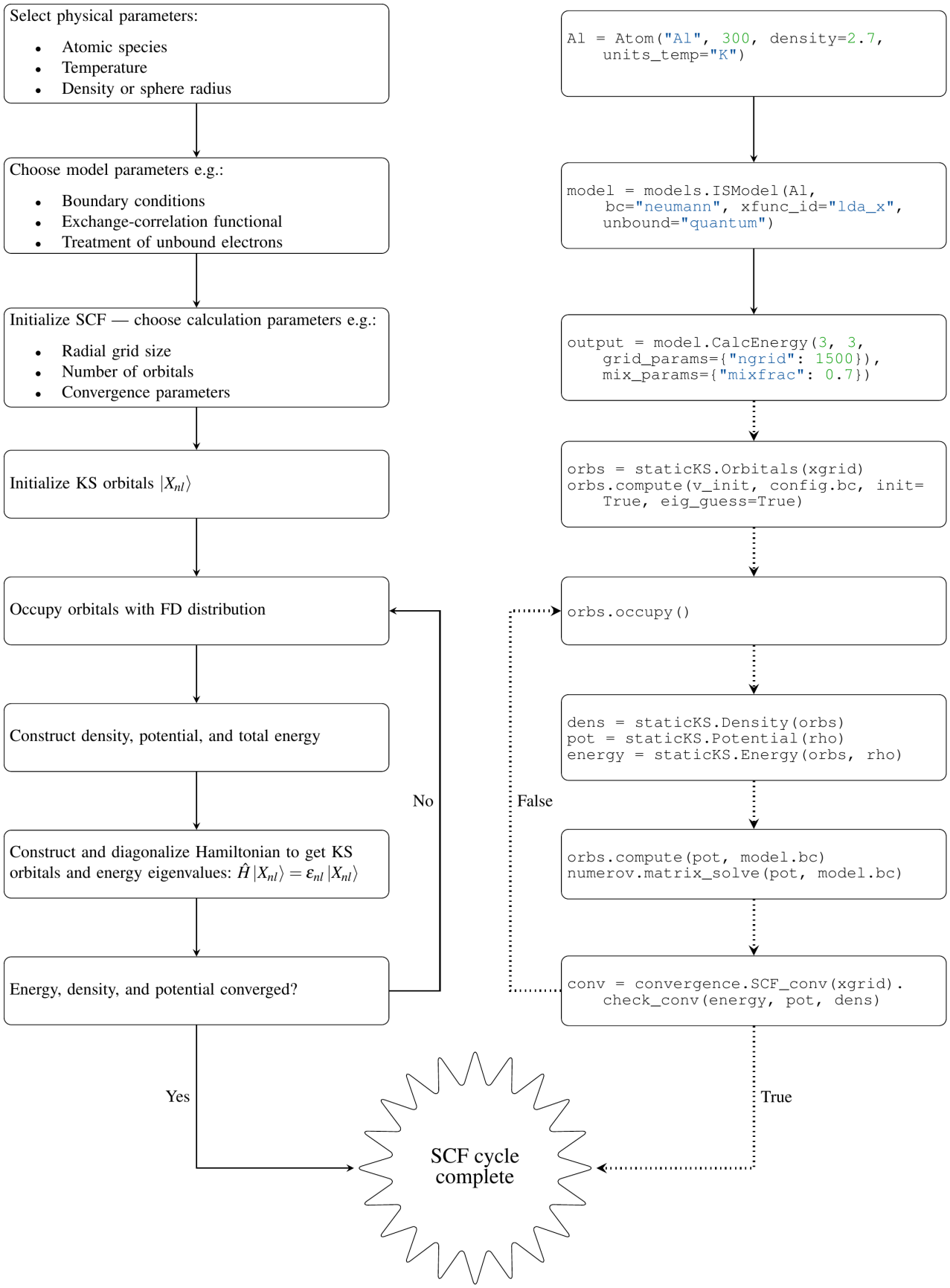


Fig. 2: *Schematic of the average-atom model set-up and the self-consistent field (SCF) cycle. On the left-hand side, the physical choices and mathematical operations that define the model and SCF cycle are shown. On the right-hand side, the (higher-order) functions and classes in atoMEC corresponding to the items on the left-hand side are shown. Some liberties are taken with the code snippets in the right-hand column of the figure to improve readability; more precisely, some non-crucial intermediate steps are not shown, and some parameters are also not shown or simplified. The dotted lines represent operations that are taken care of within the* `models.CalcEnergy` *function, but are shown nevertheless to improve understanding.*

```
Welcome to atoMEC!

Atomic information:

Atomic species               : Al
Atomic charge / weight       : 13  / 26.982
Valence electrons            : 3
Mass density                 : 2.7 g cm^-3
Voronoi sphere radius        : 2.997 Bohr / 1.586 Angstrom
Electronic temperature       : 0.00095 Ha /  0.02585 eV / 300 K
Wigner-Seitz radius          : 2.078 (Bohr)
Ionic coupling parameter     : 2.967e+04
Electron degeneracy parameter : 0.002228
```

Fig. 3: *Auto-generated print statement from calling the* `atoMEC.Atom` *object.*

```
from atoMEC import Atom
Al = Atom("Al", 300, density=2.7, units_temp="K")
```

By default, the above code automatically prints the output seen in Fig. 3. We see that the first two arguments of the `Atom` object are the chemical symbol of the element being studied, and the temperature. In addition, at least one of "density" or "radius" must be specified. In atoMEC, the default (and only permitted) units for the mass density are g cm^{-3}; *all* other input and output units in atoMEC are by default Hartree atomic units, and hence we specify "K" for Kelvin.

The information in Fig. 3 displays the chosen parameters in units commonly used in the plasma and condensed-matter physics communities, as well as some other information directly obtained from these parameters. The chemical symbol ("Al" in this case) is passed to the mendeleev library [men14] to generate this data, which is used later in the calculation.

This initial stage of the average-atom calculation, i.e. the specification of physical parameters and initialization of the `Atom` object, is shown in the top row at the top of Fig. 2.

atoMEC.models: Model parameters

After the physical parameters are set, the next stage of the average-atom calculation is to choose the model and approximations within that class of model. As discussed, so far the only class of model implemented in atoMEC is the ion-sphere model. Within this model, there are still various choices to be made by the user. In some cases, these choices make little difference to the results, but in other cases they have significant impact. The user might have some physical intuition as to which is most important, or alternatively may want to run the same physical parameters with several different model parameters to examine the effects. Some choices available in atoMEC, listed approximately in decreasing order of impact (but this can depend strongly on the system under consideration), are:

- the **boundary conditions** used to solve the KS equations;
- the treatment of the **unbound electrons**, which means those electrons not tightly bound to the nucleus, but rather delocalized over the whole atomic sphere;
- the choice of **exchange** and **correlation** functionals, the central approximations of DFT [CMSY12];
- the **spin** polarization and magnetization.

We do not discuss the theory and impact of these different choices in this paper. Rather, we direct readers to Refs. [CHKC22] and [CKC22] in which all of these choices are discussed.

In atoMEC, the ion-sphere model is controlled by the `models.ISModel` object. Continuing with our aluminum example, we choose the so-called "neumann" boundary condition,

```
Using Ion-Sphere model
Ion-sphere model parameters:

Spin-polarized               : False
Number of electrons          : 13
Exchange functional          : lda_x
Correlation functional       : lda_c_pw
Boundary condition           : neumann
Unbound electron treatment   : quantum
Shift KS potential           : True
```

Fig. 4: *Auto-generated print statement from calling the* `models.ISModel` *object.*

with a "quantum" treatment of the unbound electrons, and choose the LDA exchange functional (which is also the default). This model is set up as:

```
from atoMEC import models
model = models.ISModel(Al, bc="neumann",
             xfunc_id="lda_x", unbound="quantum")
```

By default, the above code prints the output shown in Fig. 4. The first (and only mandatory) input parameter to the `models.ISModel` object is the `Atom` object that we generated earlier. Together with the optional `spinpol` and `spinmag` parameters in the `models.ISModel` object, this sets either the total number of electrons (`spinpol=False`) or the number of electrons in each spin channel (`spinpol=True`).

The remaining information displayed in Fig. 4 shows directly the chosen model parameters, or the default values where these parameters are not specified. The exchange and correlation functionals - set by the parameters `xfunc_id` and `cfunc_id` - are passed to the LIBXC library [LSOM18] for processing. So far, only the "local density" family of approximations is available in atoMEC, and thus the default values are usually a sensible choice. For more information on exchange and correlation functionals, there are many reviews in the literature, for example Ref. [CMSY12].

This stage of the average-atom calculation, i.e. the specification of the model and the choices of approximation within that, is shown in the second row of Fig. 2.

ISModel.CalcEnergy: SCF calculation and numerical parameters

Once the physical parameters and model have been defined, the next stage in the average-atom calculation (or indeed any DFT calculation) is the SCF procedure. In atoMEC, this is invoked by the `ISModel.CalcEnergy` function. This function is called `CalcEnergy` because it finds the KS orbitals (and associated KS density) which minimize the total free energy.

Clearly, there are various mathematical and algorithmic choices in this calculation. These include, for example: the basis in which the KS orbitals and potential are represented, the algorithm used to solve the KS equations (2), and how to ensure smooth convergence of the SCF cycle. In atoMEC, the SCF procedure currently follows a single pre-determined algorithm, which we briefly review below.

In atoMEC, we represent the radial KS quantities (orbitals, density and potential) on a logarithmic grid, i.e. $x = \log(r)$. Furthermore, we make a transformation of the orbitals $P_{nl}(x) = X_{nl}(x)e^{x/2}$. Then the equations to be solved become:

$$\frac{\mathrm{d}^2 P_{nl}(x)}{\mathrm{d}x^2} - 2e^{2x}(W(x) - \varepsilon_{nl})P_{nl}(x) = 0 \tag{7}$$

$$W(x) = v_s[n](x) + \frac{1}{2}\left(l + \frac{1}{2}\right)^2 e^{-2x}. \tag{8}$$

In atoMEC, we solve the KS equations using a matrix implementation of Numerov's algorithm [PGW12]. This means we diagonalize the following equation:

$$\hat{H}\vec{P} = \vec{\varepsilon}\hat{B}\vec{P}, \text{ where} \tag{9}$$

$$\hat{H} = \hat{T} + \hat{B} + W_\mathrm{s}(\vec{x}), \tag{10}$$

$$\hat{T} = -\frac{1}{2}e^{-2\vec{x}}\hat{A}, \tag{11}$$

$$\hat{A} = \frac{\hat{I}_{-1} - 2\hat{I}_0 + \hat{I}_1}{\mathrm{d}x^2}, \text{and} \tag{12}$$

$$\hat{B} = \frac{\hat{I}_{-1} + 10\hat{I}_0 + \hat{I}_1}{12}, \tag{13}$$

In the above, $\hat{I}_{-1/0/1}$ are lower shift, identify, and upper shift matrices.

The Hamiltonian matrix $\hat{H}$ is sparse and we only seek a subset of eigenstates with lower energies: therefore there is no need to perform a full diagonalization, which scales as $\mathcal{O}(N^3)$, with N being the size of the radial grid. Instead, we use SciPy's sparse matrix diagonalization function `scipy.sparse.linalg.eigs`, which scales more efficiently and allows us to go to larger grid sizes.

After each step in the SCF cycle, the relative changes in the free energy F, density $n(r)$ and potential $v_\mathrm{s}(r)$ are computed. Specifically, the quantities computed are

$$\Delta F = \left| \frac{F^i - F^{i-1}}{F^i} \right| \tag{14}$$

$$\Delta n = \frac{\int \mathrm{d}r |n^i(r) - n^{i-1}(r)|}{\int \mathrm{d}r n^i(r)} \tag{15}$$

$$\Delta v = \frac{\int \mathrm{d}r |v_\mathrm{s}^i(r) - v_\mathrm{s}^{i-1}(r)|}{\int \mathrm{d}r v_\mathrm{s}^i(r)}. \tag{16}$$

Once all three of these metrics fall below a certain threshold, the SCF cycle is considered converged and the calculation finishes.

The SCF cycle is an example of a non-linear system and thus is prone to chaotic (non-convergent) behavior. Consequently a range of techniques have been developed to ensure convergence [SM91]. Fortunately, the tendency for calculations not to converge becomes less likely for temperatures above zero (and especially as temperatures increase). Therefore we have implemented only a simple linear mixing scheme in atoMEC. The potential used in each diagonalization step of the SCF cycle is not simply the one generated from the most recent density, but a mix of that potential and the previous one,

$$v_\mathrm{s}^{(i)}(r) = \alpha v_\mathrm{s}^i(r) + (1-\alpha) v_\mathrm{s}^{i-1}(r). \tag{17}$$

In general, a lower value of the mixing fraction α makes the SCF cycle more stable, but requires more iterations to converge. Typically a choice of $\alpha \approx 0.5$ gives a reasonable balance between speed and stability.

We can thus summarize the key parameters in an SCF calculation as follows:

- the maximum number of **eigenstates** to compute, in terms of both the principal and angular quantum numbers;
- the numerical **grid** parameters, in particular the grid size;
- the **convergence** tolerances, Eqs. (14) to (16);
- the **SCF** parameters, i.e. the mixing fraction and the maximum number of iterations.

The first three items in this list essentially control the accuracy of the calculation. In principle, for each SCF calculation — i.e. a unique set of physical and model inputs — these parameters should be independently varied until some property (such as the total free energy) is considered suitably converged with respect to that parameter. Changing the SCF parameters should not affect the final results (within the convergence tolerances), only the number of iterations in the SCF cycle.

Let us now consider an example SCF calculation, using the `Atom` and `model` objects we have already defined:

```
from atoMEC import config
config.numcores = -1 # parallelize

nmax = 3 # max value of principal quantum number
lmax = 3 # max value of angular quantum number

# run SCF calculation
scf_out = model.CalcEnergy(
 nmax,
 lmax,
 grid_params={"ngrid": 1500},
 scf_params={"mixfrac": 0.7},
 )
```

We see that the first two parameters passed to the `CalcEnergy` function are the `nmax` and `lmax` quantum numbers, which specify the number of eigenstates to compute. Precisely speaking, there is a unique Hamiltonian for each value of the angular quantum number l (and in a spin-polarized calculation, also for each spin quantum number). The sparse diagonalization routine then computes the first `nmax` eigenvalues for each Hamiltonian. In atoMEC, these diagonalizations can be run in parallel since they are independent for each value of l. This is done by setting the `config.numcores` variable to the number of cores desired (`config.numcores=-1` uses all the available cores) and handled via the joblib library [Job20].

The remaining parameters passed to the `CalcEnergy` function are optional; in the above, we have specified a grid size of 1500 points and a mixing fraction $\alpha = 0.7$. The above code automatically prints the output seen in Fig. 5. This output shows the SCF cycle and, upon completion, the breakdown of the total free energy into its various components, as well as other useful information such as the KS energy levels and their occupations.

Additionally, the output of the SCF function is a dictionary containing the `staticKS.Orbitals`, `staticKS.Density`, `staticKS.Potential` and `staticKS.Density` objects. For example, one could extract the eigenfunctions as follows:

```
orbs = scf_out["orbitals"] # orbs object
ks_eigfuncs = orbs.eigfuncs # eigenfunctions
```

The initialization of the SCF procedure is shown in the third and fourth rows of Fig. 2, with the SCF procedure itself shown in the remaining rows.

This completes the section on the code structure and algorithmic details. As discussed, with the output of an SCF calculation, there are various kinds of post-processing one can perform to obtain other properties of interest. So far in atoMEC, these are limited to the computation of the pressure (`ISModel.CalcPressure`), the electron localization function (`atoMEC.postprocess.ELFTools`) and the Kubo–Greenwood conductivity (`atoMEC.postprocess.conductivity`). We refer readers to our pre-print [CKC22] for details on how the electron localization function and the Kubo–Greenwood conductivity can be used to improve predictions of the mean ionization state.

```
Starting SCF energy calculation

iscf   E_free (Ha)    dE (1.0e-05)   dn (1.0e-04)   dv (1.0e-04)
-------------------------------------------------------------------
   0   -216.1179670      1.000e+00      9.999e-01      1.000e+00
   1   -220.6199616      2.041e-02      1.089e+00      6.059e-01
   2   -235.9883271      6.512e-02      9.080e-01      4.566e-01
   3   -241.8225927      2.413e-02      2.546e-01      1.375e-01
   4   -241.9763320      6.353e-04      4.718e-02      4.187e-02
   5   -241.9805988      1.763e-05      5.460e-03      1.517e-02
   6   -241.9809112      1.291e-06      2.035e-03      5.778e-03
   7   -241.9801006      3.350e-06      7.829e-04      2.339e-03
   8   -241.9801034      1.141e-08      2.843e-04      9.081e-04
   9   -241.9801024      4.152e-09      1.084e-04      3.559e-04
  10   -241.9801016      3.170e-09      4.137e-05      1.406e-04
  11   -241.9801013      1.603e-09      1.590e-05      5.593e-05
-------------------------------------------------------------------
SCF cycle converged

Final energies (Ha)

-------------------------------------------
Kinetic energy                :   240.0058
    orbitals                  :   240.0058
    unbound ideal approx.     :     0.0000
Electron-nuclear energy       :  -573.3573
Hartree energy                :   108.6872
Exchange-correlation energy   :   -17.3165
    exchange                  :   -16.3789
    correlation               :    -0.9376
-------------------------------------------
Total energy                  :  -241.9808
-------------------------------------------
Entropy                       :     0.0000
    orbitals                  :     0.0000
    unbound ideal approx.     :     0.0000
-------------------------------------------
Total free energy             :  -241.9808
-------------------------------------------

Chemical potential            :  -0.145
Mean ionization state         :   0.000

Orbital eigenvalues (Ha) :

     |   n=l+1 |       2 |       3
-----+---------+---------+---------
 l=0 | -54.889 |  -3.676 |  -0.169
   1 |  -2.305 |   0.058 |   1.659
   2 |   0.411 |   1.927 |   4.595


Orbital occupations (2l+1) * f_{nl} :

     |   n=l+1 |       2 |       3
-----+---------+---------+---------
 l=0 |   2.000 |   2.000 |   2.000
   1 |   6.000 |   0.000 |   0.000
   2 |   0.000 |   0.000 |   0.000
```

Fig. 5: *Auto-generated print statement from calling the* `ISModel.CalcEnergy` *function*

Case-study: Helium

In this section, we consider an application of atoMEC in the WDM regime. Helium is the second most abundant element in the universe (after hydrogen) and therefore understanding its behavior under a wide range of conditions is important for our understanding of many astrophysical processes. Of particular interest are the conditions under which helium is expected to undergo a transition from insulating to metallic behavior in the outer layers of white dwarfs, which are characterized by densities of around $1-20$ g cm^{-3} and temperatures of $10-50$ kK [PR20]. These conditions are a typical example of the WDM regime. Besides predicting the point at which the insulator-to-metallic transition occurs in the density-temperature spectrum, other properties of interest include equation-of-state data (relating pressure, density, and temperature) and electrical conductivity.

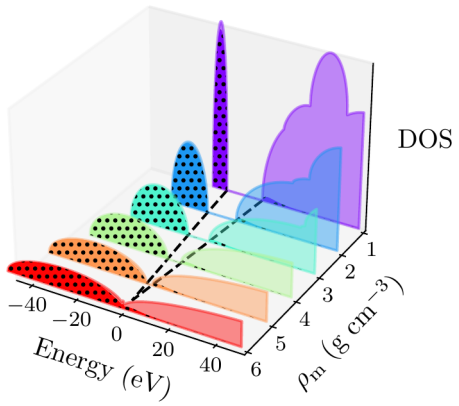


Fig. 6: *Helium density-of-states (DOS) as a function of energy, for different mass densities ρ_m, and at temperature $\tau = 50$ kK. Black dots indicate the occupations of the electrons in the permitted energy ranges. Dashed black lines indicate the band-gap (the energy gap between the insulating and conducting bands). Between 5 and 6 g cm^{-3}, the band-gap disappears.*

To calculate the insulator-to-metallic transition point, the key quantity is the electronic *band-gap*. The concept of band-structures is a complicated topic, which we try to briefly describe in layman's terms. In solids, electrons can occupy certain energy ranges — we call these the energy bands. In insulating materials, there is a gap between these energy ranges that electrons are forbidden from occupying — this is the so-called band-gap. In conducting materials, there is no such gap, and therefore electrons can conduct electricity because they can be excited into any part of the energy spectrum. Therefore, a simple method to determine the insulator-to-metallic transition is to determine the density at which the band-gap becomes zero.

In Fig. 6, we plot the density-of-states (DOS) as a function of energy, for different densities and at fixed temperature $\tau = 50$ kK. The DOS shows the energy ranges that the electrons are allowed to occupy; we also show the actual energies occupied by the electrons (according to Fermi–Dirac statistics) with the black dots. We can clearly see in this figure that the band-gap (the region where the DOS is zero) becomes smaller as a function of density. From this figure, it seems the transition from insulating to metallic state happens somewhere between 5 and 6 g cm^{-3}.

In Fig. 7, we plot the band-gap as a function of density, for a fixed temperature $\tau = 50$ kK. Visually, it appears that the relationship between band-gap and density is linear at this temperature. This is confirmed using a linear fit, which has a coefficient of determination value of almost exactly one, $R^2 = 0.9997$. Using this fit, the band-gap is predicted to close at 5.5 g cm^{-3}. Also in this figure, we show the fraction of ionized electrons, which is given by $\bar{Z}/N_e$, using Eq. (6) to calculate $\bar{Z}$, and N_e being the total electron number. The ionization fraction also relates to the conductivity of the material, because ionized electrons are not bound to any nuclei and therefore free to conduct electricity. We see that the ionization fraction mostly increases with density (excepting some strange behavior around $\rho_m = 1$ g cm^{-3}), which is further evidence of the transition from insulating to conducting behaviour with increasing density.

As a final analysis, we plot the pressure as a function of mass

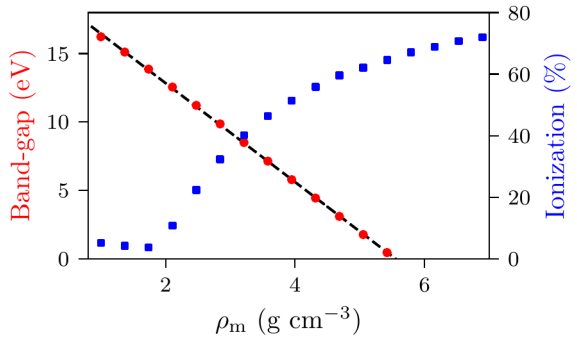


Fig. 7: *Band-gap (red circles) and ionization fraction (blue squares) for helium as a function of mass density, at temperature $\tau = 50$ kK. The relationship between the band-gap and the density appears to be linear.*

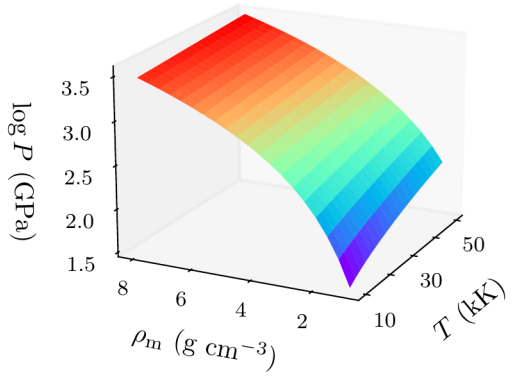


Fig. 8: *Helium pressure (logarithmic scale) as a function of mass density and temperature. The pressure increases with density and temperature (as expected), with a stronger dependence on density.*

density and temperature in Fig. 8. The pressure is given by the sum of two terms: (i) the electronic pressure, calculated using the method described in Ref. [FB19], and (ii) the ionic pressure, calculated using the ideal gas law. We observe that the pressure increases with both density and temperature, which is the expected behavior. Under these conditions, the density dependence is much stronger, especially for higher densities.

The code required to generate the above results and plots can be found in this repository.

Conclusions and future work

In this paper, we have presented atoMEC: an average-atom Python code for studying materials under extreme conditions. The open-source nature of atoMEC, and the choice to use (pure) Python as the programming language, is designed to improve the accessibility of average-atom models.

We gave significant attention to the code structure in this paper, and tried as much as possible to connect the functions and objects in the code with the underlying theory. We hope that this not only improves atoMEC from a user perspective, but also facilitates new contributions from the wider average-atom, WDM and scientific Python communities. Another aim of the paper was to communicate how atoMEC benefits from a strong ecosystem of open-source scientific libraries — especially the Python libraries NumPy, SciPy, joblib and mendeleev, as well as LIBXC.

We finish this paper by emphasizing that atoMEC is still in the early stages of development, and there are many opportunities to improve and extend the code. These include, for example:

- adding new average-atom models, and different approximations to the existing `models.ISModel` model;
- optimizing the code, in particular the routines in the `numerov` module;
- adding new postprocessing functionality, for example to compute structure factors;
- improving the structure and design choices of the code.

Of course, these are just a snapshot of the avenues for future development in atoMEC. We are open to contributions in these areas and many more besides.

Acknowledgements

This work was partly funded by the Center for Advanced Systems Understanding (CASUS) which is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

References

[BDM+20] M. Bonitz, T. Dornheim, Zh. A. Moldabekov, S. Zhang, P. Hamann, H. Kählert, A. Filinov, K. Ramakrishna, and J. Vorberger. *Ab initio* simulation of warm dense matter. *Phys. Plasmas*, 27(4):042710, 2020. doi:10.1063/1.5143225.

[BNR13] Roi Baer, Daniel Neuhauser, and Eran Rabani. Self-averaging stochastic Kohn-Sham density-functional theory. *Phys. Rev. Lett.*, 111:106402, Sep 2013. doi:10.1103/PhysRevLett.111.106402.

[BVL+17] Felix Brockherde, Leslie Vogt, Li Li, Mark E. Tuckerman, Kieron Burke, and Klaus-Robert Müller. Bypassing the Kohn-Sham equations with machine learning. *Nature Communications*, 8(1):872, Oct 2017. doi:10.1038/s41467-017-00839-3.

[CHKC22] T. J. Callow, S. B. Hansen, E. Kraisler, and A. Cangi. First-principles derivation and properties of density-functional average-atom models. *Phys. Rev. Research*, 4:023055, Apr 2022. doi:10.1103/PhysRevResearch.4.023055.

[CKC22] Timothy J. Callow, Eli Kraisler, and Attila Cangi. Accurate and efficient computation of mean ionization states with an average-atom Kubo-Greenwood approach, 2022. doi:10.48550/ARXIV.2203.05863.

[CKTS+21] Timothy Callow, Daniel Kotik, Ekaterina Tsvetoslavova Stankulova, Eli Kraisler, and Attila Cangi. atomec, August 2021. If you use this software, please cite it using these metadata. doi:10.5281/zenodo.5205719.

[CMSY12] Aron J. Cohen, Paula Mori-Sánchez, and Weitao Yang. Challenges for density functional theory. *Chemical Reviews*, 112(1):289–320, 2012. doi:10.1021/cr200107z.

[CRNB18] Yael Cytter, Eran Rabani, Daniel Neuhauser, and Roi Baer. Stochastic density functional theory at finite temperatures. *Phys. Rev. B*, 97:115207, Mar 2018. doi:10.1103/PhysRevB.97.115207.

[DGB18] Tobias Dornheim, Simon Groth, and Michael Bonitz. The uniform electron gas at warm dense matter conditions. *Phys. Rep.*, 744:1 – 86, 2018. doi:10.1016/j.physrep.2018.04.001.

[EFP+21] J. A. Ellis, L. Fiedler, G. A. Popoola, N. A. Modine, J. A. Stephens, A. P. Thompson, A. Cangi, and S. Rajamanickam. Accelerating finite-temperature kohn-sham density functional theory with deep neural networks. *Phys. Rev. B*, 104:035120, Jul 2021. doi:10.1103/PhysRevB.104.035120.

[FB19] Gérald Faussurier and Christophe Blancard. Pressure in warm and hot dense matter using the average-atom model. *Phys. Rev. E*, 99:053201, May 2019. doi:10.1103/PhysRevE.99.053201.

[GDRT14] Frank Graziani, Michael P Desjarlais, Ronald Redmer, and Samuel B Trickey. *Frontiers and challenges in warm dense matter*, volume 96. Springer Science & Business, 2014. doi:10.1007/978-3-319-04912-0.

[GFG+16] S H Glenzer, L B Fletcher, E Galtier, B Nagler, R Alonso-Mori, B Barbrel, S B Brown, D A Chapman, Z Chen, C B Curry, F Fiuza, E Gamboa, M Gauthier, D O Gericke, A Gleason, S Goede, E Granados, P Heimann, J Kim, D Kraus, M J MacDonald, A J Mackinnon, R Mishra, A Ravasio, C Roedel, P Sperling, W Schumaker, Y Y Tsui, J Vorberger, U Zastrau, A Fry, W E White, J B Hasting, and H J Lee. Matter under extreme conditions experiments at the Linac Coherent Light Source. *J. Phys. B*, 49(9):092001, apr 2016. doi:10.1088/0953-4075/49/9/092001.

[HK64] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136(3B):B864–B871, Nov 1964. doi:10.1103/PhysRev.136.B864.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[HRD08] Bastian Holst, Ronald Redmer, and Michael P. Desjarlais. Thermophysical properties of warm dense hydrogen using quantum molecular dynamics simulations. *Phys. Rev. B*, 77:184201, May 2008. doi:10.1103/PhysRevB.77.184201.

[JFC+13] Weile Jia, Jiyun Fu, Zongyan Cao, Long Wang, Xuebin Chi, Weiguo Gao, and Lin-Wang Wang. Fast plane wave density functional theory molecular dynamics calculations on multi-GPU machines. *Journal of Computational Physics*, 251:102–115, 2013. doi:10.1016/j.jcp.2013.05.005.

[Job20] Joblib Development Team. Joblib: running Python functions as pipeline jobs. https://joblib.readthedocs.io/, 2020.

[KDF+11] A. L. Kritcher, T. Döppner, C. Fortmann, T. Ma, O. L. Landen, R. Wallace, and S. H. Glenzer. In-Flight Measurements of Capsule Shell Adiabats in Laser-Driven Implosions. *Phys. Rev. Lett.*, 107:015002, Jul 2011. doi:10.1103/PhysRevLett.107.015002.

[Koh99] W. Kohn. Nobel lecture: Electronic structure of matter—wave functions and density functionals. *Rev. Mod. Phys.*, 71:1253–1266, 10 1999. doi:10.1103/RevModPhys.71.1253.

[KS65] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1133–A1138, Nov 1965. doi:10.1103/PhysRev.140.A1133.

[LSOM18] Susi Lehtola, Conrad Steigemann, Micael J.T. Oliveira, and Miguel A.L. Marques. Recent developments in LIBXC — A comprehensive library of functionals for density functional theory. *SoftwareX*, 7:1–5, 2018. doi:10.1016/j.softx.2017.11.002.

[MED11] Stefan Maintz, Bernhard Eck, and Richard Dronskowski. Speeding up plane-wave electronic-structure calculations using graphics-processing units. *Computer Physics Communications*, 182(7):1421–1427, 2011. doi:10.1016/j.cpc.2011.03.010.

[men14] mendeleev – A Python resource for properties of chemical elements, ions and isotopes, ver. 0.9.0. https://github.com/lmmentel/mendeleev, 2014.

[Mer65] N. David Mermin. Thermal properties of the inhomogeneous electron gas. *Phys. Rev.*, 137:A1441–A1443, Mar 1965. doi:10.1103/PhysRev.137.A1441.

[PGW12] Mohandas Pillai, Joshua Goglio, and Thad G. Walker. Matrix numerov method for solving schrödinger's equation. *American Journal of Physics*, 80(11):1017–1019, 2012. doi:10.1119/1.4748813.

[PPF+11] S. Pittalis, C. R. Proetto, A. Floris, A. Sanna, C. Bersier, K. Burke, and E. K. U. Gross. Exact conditions in finite-temperature density-functional theory. *Phys. Rev. Lett.*, 107:163001, Oct 2011. doi:10.1103/PhysRevLett.107.163001.

[PR20] Martin Preising and Ronald Redmer. Metallization of dense fluid helium from ab initio simulations. *Phys. Rev. B*, 102:224107, Dec 2020. doi:10.1103/PhysRevB.102.224107.

[Roz91] Balazs F. Rozsnyai. Photoabsorption in hot plasmas based on the ion-sphere and ion-correlation models. *Phys. Rev. A*, 43:3035–3042, Mar 1991. doi:10.1103/PhysRevA.43.3035.

[SM91] H. B. Schlegel and J. J. W. McDouall. *Do You Have SCF Stability and Convergence Problems?*, pages 167–185. Springer Netherlands, Dordrecht, 1991. doi:10.1007/978-94-011-3262-6_2.

[SPS+14] A. N. Souza, D. J. Perkins, C. E. Starrett, D. Saumon, and S. B. Hansen. Predictions of x-ray scattering spectra for warm dense matter. *Phys. Rev. E*, 89:023108, Feb 2014. doi:10.1103/PhysRevE.89.023108.

[SRH+12] John C. Snyder, Matthias Rupp, Katja Hansen, Klaus-Robert Müller, and Kieron Burke. Finding density functionals with machine learning. *Phys. Rev. Lett.*, 108:253002, Jun 2012. doi:10.1103/PhysRevLett.108.253002.

[SS14] C.E. Starrett and D. Saumon. A simple method for determining the ionic structure of warm dense matter. *High Energy Density Physics*, 10:35–42, 2014. doi:10.1016/j.hedp.2013.12.001.

[Sta16] C.E. Starrett. Kubo–Greenwood approach to conductivity in dense plasmas with average atom models. *High Energy Density Physics*, 19:58–64, 2016. doi:10.1016/j.hedp.2016.04.001.

[STJ+14] Sang-Kil Son, Robert Thiele, Zoltan Jurek, Beata Ziaja, and Robin Santra. Quantum-mechanical calculation of ionization-potential lowering in dense plasmas. *Phys. Rev. X*, 4:031004, Jul 2014. doi:10.1103/PhysRevX.4.031004.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

Automatic random variate generation in Python

Christoph Baumgarten[‡*], Tirth Patel

Abstract—The generation of random variates is an important tool that is required in many applications. Various software programs or packages contain generators for standard distributions like the normal, exponential or Gamma, e.g., the programming language R and the packages SciPy and NumPy in Python. However, it is not uncommon that sampling from new/non-standard distributions is required. Instead of deriving specific generators in such situations, so-called automatic or black-box methods have been developed. These allow the user to generate random variates from fairly large classes of distributions by only specifying some properties of the distributions (e.g. the density and/or cumulative distribution function). In this note, we describe the implementation of such methods from the C library UNU.RAN in the Python package SciPy and provide a brief overview of the functionality.

Index Terms—numerical inversion, generation of random variates

Introduction

The generation of random variates is an important tool that is required in many applications. Various software programs or packages contain generators for standard distributions, e.g., R ([R C21]) and SciPy ([VGO+20]) and NumPy ([HMvdW+20]) in Python. Standard references for these algorithms are the books [Dev86], [Dag88], [Gen03], and [Knu14]. An interested reader will find many references to the vast existing literature in these works. While relying on general methods such as the rejection principle, the algorithms for well-known distributions are often specifically designed for a particular distribution. This is also the case in the module `stats` in SciPy that contains more than 100 distributions and the module `random` in NumPy with more than 30 distributions. However, there are also so-called automatic or black-box methods for sampling from large classes of distributions with a single piece of code. For such algorithms, information about the distribution such as the density, potentially together with its derivative, the cumulative distribution function (CDF), and/or the mode must be provided. See [HLD04] for a comprehensive overview of these methods. Although the development of such methods was originally motivated to generate variates from non-standard distributions, these universal methods have advantages that make their usage attractive even for sampling from standard distributions. We mention some of the important properties (see [LH00], [HLD04], [DHL10]):

- The algorithms can be used to sample from truncated distributions.
- For inversion methods, the structural properties of the underlying uniform random number generator are preserved and the numerical accuracy of the methods can be controlled by a parameter. Therefore, inversion is usually the only method applied for simulations using quasi-Monte Carlo (QMC) methods.
- Depending on the use case, one can choose between a fast setup with slow marginal generation time and vice versa.

The latter point is important depending on the use case: if a large number of samples is required for a given distribution with fixed shape parameters, a slower setup that only has to be run once can be accepted if the marginal generation times are low. If small to moderate samples sizes are required for many different shape parameters, then it is important to have a fast setup. The former situation is referred to as the fixed-parameter case and the latter as the varying parameter case.

Implementations of various methods are available in the C library UNU.RAN ([HL07]) and in the associated R package `Runuran` (https://cran.r-project.org/web/packages/Runuran/index.html, [TL03]). The aim of this note is to introduce the Python implementation in the SciPy package that makes some of the key methods in UNU.RAN available to Python users in SciPy 1.8.0. These general tools can be seen as a complement to the existing specific sampling methods: they might lead to better performance in specific situations compared to the existing generators, e.g., if a very large number of samples are required for a fixed parameter of a distribution or if the implemented sampling method relies on a slow default that is based on numerical inversion of the CDF. For advanced users, they also offer various options that allow to fine-tune the generators (e.g., to control the time needed for the setup step).

Automatic algorithms in SciPy

Many of the automatic algorithms described in [HLD04] and [DHL10] are implemented in the ANSI C library, UNU.RAN (Universal Non-Uniform RANdom variate generators). Our goal was to provide a Python interface to the most important methods from UNU.RAN to generate univariate discrete and continuous non-uniform random variates. The following generators have been implemented in SciPy 1.8.0:

- `TransformedDensityRejection`: Transformed Density Rejection (TDR) ([HÖ95], [GW92])
- `NumericalInverseHermite`: Hermite interpolation based INVersion of CDF (HINV) ([HL03])
- `NumericalInversePolynomial`: Polynomial interpolation based INVersion of CDF (PINV) ([DHL10])

* *Corresponding author: christoph.baumgarten@gmail.com*
‡ *Unaffiliated*

- `SimpleRatioUniforms`: Simple Ratio-Of-Uniforms (SROU) ([Ley01], [Ley03])
- `DiscreteGuideTable`: (Discrete) Guide Table method (DGT) ([CA74])
- `DiscreteAliasUrn`: (Discrete) Alias-Urn method (DAU) ([Wal77])

Before describing the implementation in SciPy in Section scipy_impl, we give a short introduction to random variate generation in Section intro_rv_gen.

A very brief introduction to random variate generation

It is well-known that random variates can be generated by inversion of the CDF F of a distribution: if U is a uniform random number on $(0,1)$, $X := F^{-1}(U)$ is distributed according to F. Unfortunately, the inverse CDF can only be expressed in closed form for very few distributions, e.g., the exponential or Cauchy distribution. If this is not the case, one needs to rely on implementations of special functions to compute the inverse CDF for standard distributions like the normal, Gamma or beta distributions or numerical methods for inverting the CDF are required. Such procedures, however, have the disadvantage that they may be slow or inaccurate, and developing fast and robust inversion algorithms such as HINV and PINV is a non-trivial task. HINV relies on Hermite interpolation of the inverse CDF and requires the CDF and PDF as an input. PINV only requires the PDF. The algorithm then computes the CDF via adaptive Gauss-Lobatto integration and an approximation of the inverse CDF using Newton's polynomial interpolation. Note that an approximation of the inverse CDF can be achieved by interpolating the points $(F(x_i), x_i)$ for points x_i in the domain of F, i.e., no evaluation of the inverse CDF is required.

For discrete distributions, F is a step-function. To compute the inverse CDF $F^{-1}(U)$, the simplest idea would be to apply sequential search: if X takes values $0,1,2,\ldots$ with probabilities $p_0, p_1, p_2, \ldots$, start with $j = 0$ and keep incrementing j until $F(j) = p_0 + \cdots + p_j \geq U$. When the search terminates, $X = j = F^{-1}(U)$. Clearly, this approach is generally very slow and more efficient methods have been developed: if X takes L distinct values, DGT realizes very fast inversion using so-called guide tables / hash tables to find the index j. In contrast DAU is not an inversion method but uses the alias method, i.e., tables are precomputed to write X as an equi-probable mixture of L two-point distributions (the alias values).

The rejection method has been suggested in [VN51]. In its simplest form, assume that f is a bounded density on $[a,b]$, i.e., $f(x) \leq M$ for all $x \in [a,b]$. Sample two independent uniform random variates on U on $[0,1]$ and V on $[a,b]$ until $M \cdot U \leq f(V)$. Note that the accepted points (U,V) are uniformly distributed in the region between the x-axis and the graph of the PDF. Hence, $X := V$ has the desired distribution f. This is a special case of the general version: if f, g are two densities on an interval J such that $f(x) \leq c \cdot g(x)$ for all $x \in J$ and a constant $c \geq 1$, sample U uniformly distributed on $[0,1]$ and X distributed according to g until $c \cdot U \cdot g(X) \leq f(X)$. Then X has the desired distribution f. It can be shown that the expected number of iterations before the acceptance condition is met is equal to c. Hence, the main challenge is to find hat functions g for which c is small and from which random variates can be generated efficiently. TDR solves this problem by applying a transformation T to the density such that $x \mapsto T(f(x))$ is concave. A hat function can then be found by computing tangents at suitable design points. Note that by its nature any rejection method requires not always the same number of uniform variates to generate one non-uniform variate; this makes the use of QMC and of some variance reduction methods more difficult or impossible. On the other hand, rejection is often the fastest choice for the varying parameter case.

The Ratio-Of-Uniforms method (ROU, [KM77]) is another general method that relies on rejection. The underlying principle is that if (U,V) is uniformly distributed on the set $A_f := \{(u,v) : 0 < v \leq \sqrt{f(u/v)}, a < u/v < b\}$ where f is a PDF with support (a,b), then $X := U/V$ follows a distribution according to f. In general, it is not possible to sample uniform values on A_f directly. However, if $A_f \subset R := [u_-, u_+] \times [0, v_+]$ for finite constants u_-, u_+, v_+, one can apply the rejection method: generate uniform values (U,V) on the bounding rectangle R until $(U,V) \in A_f$ and return $X = U/V$. Automatic methods relying on the ROU method such as SROU and automatic ROU ([Ley00]) need a setup step to find a suitable region $S \in \mathbb{R}^2$ such that $A_f \subset S$ and such that one can generate (U,V) uniformly on S efficiently.

Description of the SciPy interface

SciPy provides an object-oriented API to UNU.RAN's methods. To initialize a generator, two steps are required:

1) creating a distribution class and object,
2) initializing the generator itself.

In step 1, a distributions object must be created that implements required methods (e.g., `pdf`, `cdf`). This can either be a custom object or a distribution object from the classes `rv_continuous` or `rv_discrete` in SciPy. Once the generator is initialized from the distribution object, it provides a `rvs` method to sample random variates from the given distribution. It also provides a `ppf` method that approximates the inverse CDF if the initialized generator uses an inversion method. The following example illustrates how to initialize the `NumericalInversePolynomial` (PINV) generator for the standard normal distribution:

```
import numpy as np
from scipy.stats import sampling
from math import exp

# create a distribution class with implementation
# of the PDF. Note that the normalization constant
# is not required
class StandardNormal:
    def pdf(self, x):
        return exp(-0.5 * x**2)

# create a distribution object and initialize the
# generator
dist = StandardNormal()
rng = sampling.NumericalInversePolynomial(dist)

# sample 100,000 random variates from the given
# distribution
rvs = rng.rvs(100000)
```

As `NumericalInversePolynomial` generator uses an inversion method, it also provides a `ppf` method that approximates the inverse CDF:

```
# evaluate the approximate PPF at a few points
ppf = rng.ppf([0.1, 0.5, 0.9])
```

It is also easy to sample from a truncated distribution by passing a `domain` argument to the constructor of the generator. For example, to sample from truncated normal distribution:

```
# truncate the distribution by passing a
# `domain` argument
rng = sampling.NumericalInversePolynomial(
   dist, domain=(-1, 1)
)
```

While the default options of the generators should work well in many situations, we point out that there are various parameters that the user can modify, e.g., to provide further information about the distribution (such as `mode` or `center`) or to control the numerical accuracy of the approximated PPF. (`u_resolution`). Details can be found in the SciPy documentation https://docs.scipy.org/doc/scipy/reference/. The above code can easily be generalized to sample from parametrized distributions using instance attributes in the distribution class. For example, to sample from the gamma distribution with shape parameter `alpha`, we can create the distribution class with parameters as instance attributes:

```
class Gamma:
    def __init__(self, alpha):
        self.alpha = alpha

    def pdf(self, x):
        return x**(self.alpha-1) * exp(-x)

    def support(self):
        return 0, np.inf

# initialize a distribution object with varying
# parameters
dist1 = Gamma(2)
dist2 = Gamma(3)

# initialize a generator for each distribution
rng1 = sampling.NumericalInversePolynomial(dist1)
rng2 = sampling.NumericalInversePolynomial(dist2)
```

In the above example, the `support` method is used to set the domain of the distribution. This can alternatively be done by passing a `domain` parameter to the constructor.

In addition to continuous distribution, two UNU.RAN methods have been added in SciPy to sample from discrete distributions. In this case, the distribution can be either be represented using a probability vector (which is passed to the constructor as a Python list or NumPy array) or a Python object with the implementation of the probability mass function. In the latter case, a finite domain must be passed to the constructor or the object should implement the `support` method[1].

```
# Probability vector to represent a discrete
# distribution. Note that the probability vector
# need not be vectorized
pv = [0.1, 9.0, 2.9, 3.4, 0.3]

# PCG64 uniform RNG with seed 123
urng = np.random.default_rng(123)
rng = sampling.DiscreteAliasUrn(
   pv, random_state=urng
)

# sample from the given discrete distribution
rvs = rng.rvs(100000)
```

Underlying uniform pseudo-random number generators

NumPy provides several generators for uniform pseudo-random numbers[2]. It is highly recommended to use NumPy's default random number generator `np.random.PCG64` for better speed and performance, see [O'N14] and https://numpy.org/doc/stable/reference/random/bit_generators/index.html. To change the uniform random number generator, a `random_state` parameter can be passed as shown in the example below:

```
# 64-bit PCG random number generator in NumPy
urng = np.random.Generator(np.random.PCG64())
# The above line can also be replaced by:
# ``urng = np.random.default_rng()``
# as PCG64 is the default generator starting
# from NumPy 1.19.0

# change the uniform random number generator by
# passing the `random_state` argument
rng = sampling.NumericalInversePolynomial(
   dist, random_state=urng
)
```

We also point out that the PPF of inversion methods can be applied to sequences of quasi-random numbers. SciPy provides different sequences in its QMC module (`scipy.stats.qmc`).

`NumericalInverseHermite` provides a `qrvs` method which generates random variates using QMC methods present in SciPy (`scipy.stats.qmc`) as uniform random number generators[3]. The next example illustrates how to use `qrvs` with a generator created directly from a SciPy distribution object.

```
from scipy import stats
from scipy.stats import qmc

# 1D Halton sequence generator.
qrng = qmc.Halton(d=1)

rng = sampling.NumericalInverseHermite(stats.norm())

# generate quasi random numbers using the Halton
# sequence as uniform variates
qrvs = rng.qrvs(size=100, qmc_engine=qrng)
```

Benchmarking

To analyze the performance of the implementation, we tested the methods applied to several standard distributions against the generators in NumPy and the original UNU.RAN C library. In addition, we selected one non-standard distribution to demonstrate that substantial reductions in the runtime can be achieved compared to other implementations. All the benchmarks were carried out using NumPy 1.22.4 and SciPy 1.8.1 running in a single core on Ubuntu 20.04.3 LTS with Intel(R) Core(TM) i7-8750H CPU (2.20GHz clock speed, 16GB RAM). We run the benchmarks with NumPy's MT19937 (Mersenne Twister) and PCG64 random number generators (`np.random.MT19937` and `np.random.PCG64`) in Python and use NumPy's C implementation of MT19937 in the UNU.RAN C benchmarks. As explained above, the use of PCG64 is recommended, and MT19937 is only included to compare the speed of the Python implementation and the C library by relying on the same uniform number generator (i.e., differences in the performance of the uniform number generation are not taken into account). The code for all the benchmarks can be found on https://github.com/tirthasheshpatel/unuran_benchmarks.

The methods used in NumPy to generate normal, gamma, and beta random variates are:

- the ziggurat algorithm ([MT00b]) to sample from the standard normal distribution,

1. Support for discrete distributions with infinite domain hasn't been added yet.

2. By default, NumPy's legacy random number generator, MT19937 (`np.random.RandomState()`) is used as the uniform random number generator for consistency with the stats module in SciPy.

3. In SciPy 1.9.0, `qrvs` will be added to `NumericalInversePolynomial`.

- the rejection algorithms in Chapter XII.2.6 in [Dev86] if $\alpha < 1$ and in [MT00a] if $\alpha > 1$ for the Gamma distribution,
- Johnk's algorithm ([Jöh64], Section IX.3.5 in [Dev86]) if $\max\{\alpha, \beta\} \leq 1$, otherwise a ratio of two Gamma variates with shape parameter α and β (see Section IX.4.1 in [Dev86]) for the beta distribution.

Benchmarking against the normal, gamma, and beta distributions

Table 1 compares the performance for the standard normal, Gamma and beta distributions. We recall that the density of the Gamma distribution with shape parameter $a > 0$ is given by $x \in (0,\infty) \mapsto x^{a-1}e^{-x}$ and the density of the beta distribution with shape parameters $\alpha, \beta > 0$ is given by $x \in (0,1) \mapsto \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$ where $\Gamma(\cdot)$ and $B(\cdot,\cdot)$ are the Gamma and beta functions. The results are reported in Table 1.

We summarize our main observations:

1) The setup step in Python is substantially slower than in C due to expensive Python callbacks, especially for PINV and HINV. However, the time taken for the setup is low compared to the sampling time if large samples are drawn. Note that as expected, SROU has a very fast setup such that this method is suitable for the varying parameter case.
2) The sampling time in Python is slightly higher than in C for the MT19937 random number generator. If the recommended PCG64 generator is used, the sampling time in Python is slightly lower. The only exception is SROU: due to Python callbacks, the performance is substantially slower than in C. However, as the main advantage of SROU is the fast setup time, the main use case is the varying parameter case (i.e., the method is not supposed to be used to generate large samples).
3) PINV, HINV, and TDR are at most about 2x slower than the specialized NumPy implementation for the normal distribution. For the Gamma and beta distribution, they even perform better for some of the chosen shape parameters. These results underline the strong performance of these black-box approaches even for standard distributions.
4) While the application of PINV requires bounded densities, no issues are encountered for $\alpha = 0.05$ since the unbounded part is cut off by the algorithm. However, the setup can fail for very small values of α.

Benchmarking against a non-standard distribution

We benchmark the performance of PINV to sample from the generalized normal distribution ([Sub23]) whose density is given by $x \in (-\infty,\infty) \mapsto \frac{pe^{-|x|^p}}{2\Gamma(1/p)}$ against the method proposed in [NP09] and against the implementation in SciPy's `gennorm` distribution. The approach in [NP09] relies on transforming Gamma variates to the generalized normal distribution whereas SciPy relies on computing the inverse of CDF of the Gamma distribution (https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.gammainccinv.html). The results for different values of p are shown in Table 2.

PINV is usually about twice as fast than the specialized method and about 15-150 times faster than SciPy's implementation[4]. We also found an R package `pgnorm` (https://cran.r-project.org/web/packages/pgnorm/) that implements various approaches from [KR13]. In that case, PINV is usually about 70-200 times faster. This clearly shows the benefit of using a black-box algorithm.

Conclusion

The interface to UNU.RAN in SciPy provides easy access to different algorithms for non-uniform variate generation for large classes of univariate continuous and discrete distributions. We have shown that the methods are easy to use and that the algorithms perform very well both for standard and non-standard distributions. A comprehensive documentation suite, a tutorial and many examples are available at https://docs.scipy.org/doc/scipy/reference/stats.sampling.html and https://docs.scipy.org/doc/scipy/tutorial/stats/sampling.html. Various methods have been implemented in SciPy, and if specific use cases require additional functionality from UNU.RAN, the methods can easily be added to SciPy given the flexible framework that has been developed. Another area of further development is to better integrate SciPy's QMC generators for the inversion methods.

Finally, we point out that other sampling methods like Markov Chain Monte Carlo and copula methods are not part of SciPy. Relevant Python packages in that context are PyMC ([PHF10]), PyStan relying on Stan ([Tea21]), Copulas (https://sdv.dev/Copulas/) and PyCopula (https://blent-ai.github.io/pycopula/).

Acknowledgments

The authors wish to thank Wolfgang Hörmann and Josef Leydold for agreeing to publish the library under a BSD license and for helpful feedback on the implementation and this note. In addition, we thank Ralf Gommers, Matt Haberland, Nicholas McKibben, Pamphile Roy, and Kai Striega for their code contributions, reviews, and helpful suggestions. The second author was supported by the Google Summer of Code 2021 program[5].

References

[CA74] Hui-Chuan Chen and Yoshinori Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974. doi:10.1080/05695557408974949.

[Dag88] John Dagpunar. *Principles of random variate generation*. Oxford University Press, USA, 1988.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986. doi:10.1007/978-1-4613-8643-8.

[DHL10] Gerhard Derflinger, Wolfgang Hörmann, and Josef Leydold. Random variate generation by numerical inversion when only the density is known. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):1–25, 2010. doi:10.1145/1842722.1842723.

[Gen03] James E Gentle. *Random number generation and Monte Carlo methods*, volume 381. Springer, 2003. doi:10.1007/b97336.

[GW92] Walter R Gilks and Pascal Wild. Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(2):337–348, 1992. doi:10.2307/2347565.

[Hö95] Wolfgang Hörmann. A rejection technique for sampling from T-concave distributions. *ACM Trans. Math. Softw.*, 21(2):182–193, 1995. doi:10.1145/203082.203089.

[HL03] Wolfgang Hörmann and Josef Leydold. Continuous random variate generation by fast numerical inversion. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4):347–362, 2003. doi:10.1145/945511.945517.

4. In SciPy 1.9.0, the speed will be improved by implementing the method from [NP09]

5. https://summerofcode.withgoogle.com/projects/#5912428874825728

Distribution	Method	Python			C	
		Setup	Sampling (PCG64)	Sampling (MT19937)	Setup	Sampling (MT19937)
Standard normal	PINV	4.6	29.6	36.5	0.27	32.4
	HINV	2.5	33.7	40.9	0.38	36.8
	TDR	0.2	37.3	47.8	0.02	41.4
	SROU	8.7 µs	2510	2160	0.5 µs	232
	NumPy	-	17.6	22.4	-	-
Gamma(0.05)	PINV	196.0	29.8	37.2	37.9	32.5
	HINV	24.5	36.1	43.8	1.9	40.7
	NumPy	-	55.0	68.1	-	-
Gamma(0.5)	PINV	16.5	31.2	38.6	2.0	34.5
	HINV	4.9	34.2	41.7	0.6	37.9
	NumPy	-	86.4	99.2	-	-
Gamma(3.0)	PINV	5.3	30.8	38.7	0.5	34.6
	HINV	5.3	33	40.6	0.4	36.8
	TDR	0.2	38.8	49.6	0.03	44
	NumPy	-	36.5	47.1	-	-
Beta(0.5, 0.5)	PINV	21.4	33.1	39.9	2.4	37.3
	HINV	2.1	38.4	45.3	0.2	42
	NumPy	-	101	112	-	-
Beta(0.5, 1.0)	HINV	0.2	37	44.3	0.01	41.1
	NumPy	-	125	138	-	-
Beta(1.3, 1.2)	PINV	15.7	30.5	37.2	1.7	34.3
	HINV	4.1	33.4	40.8	0.4	37.1
	TDR	0.2	46.8	57.8	0.03	45
	NumPy	-	74.3	97	-	-
Beta(3.0, 2.0)	PINV	9.7	30.2	38.2	0.9	33.8
	HINV	5.8	33.7	41.2	0.4	37.4
	TDR	0.2	42.8	52.8	0.02	44
	NumPy	-	72.6	92.8	-	-

TABLE 1

Average time taken (reported in milliseconds, unless mentioned otherwise) to sample 1 million random variates from the standard normal distribution. The mean is computed over 7 iterations. Standard deviations are not reported as they were very small (less than 1% of the mean in the large majority of cases). Note that not all methods can always be applied, e.g., TDR cannot be applied to the Gamma distribution if $a < 1$ since the PDF is not log-concave in that case. As NumPy uses rejection algorithms with precomputed constants, no setup time is reported.

p	0.25	0.45	0.75	1	1.5	2	5	8
Nardon and Pianca (2009)	100	101	101	45	148	120	128	122
SciPy's `gennorm` distribution	832	1000	1110	559	5240	6720	6230	5950
Python (PINV Method, PCG64 urng)	50	47	45	41	40	37	38	38

TABLE 2

Comparing SciPy's implementation and a specialized method against PINV to sample 1 million variates from the generalized normal distribution for different values of the parameter p. Time reported in milliseconds. The mean is computer over 7 iterations.

[HL07] Wolfgang Hörmann and Josef Leydold. UNU.RAN - Universal Non-Uniform RANdom number generators, 2007. https://statmath.wu.ac.at/unuran/doc.html.

[HLD04] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. *Automatic nonuniform random variate generation*. Springer, 2004. doi:10.1007/978-3-662-05946-3.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.

[Jöh64] MD Jöhnk. Erzeugung von betaverteilten und gammaverteilten Zufallszahlen. *Metrika*, 8(1):5–15, 1964. doi:10.1007/bf02613706.

[KM77] Albert J Kinderman and John F Monahan. Computer generation of random variables using the ratio of uniform deviates. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):257–260, 1977. doi:10.1145/355744.355750.

[Knu14] Donald E Knuth. *The Art of Computer Programming, Volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014. doi:10.2307/2317055.

[KR13] Steve Kalke and W-D Richter. Simulation of the p-generalized Gaussian distribution. *Journal of Statistical Computation and Simulation*, 83(4):641–667, 2013. doi:10.1080/00949655.2011.631187.

[Ley00] Josef Leydold. Automatic sampling with the ratio-of-uniforms method. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):78–98, 2000. doi:10.1145/347837.347863.

[Ley01] Josef Leydold. A simple universal generator for continuous and discrete univariate T-concave distributions. *ACM Transactions on Mathematical Software (TOMS)*, 27(1):66–82, 2001. doi:10.1145/382043.382322.

[Ley03] Josef Leydold. Short universal generators via generalized ratio-of-uniforms method. *Mathematics of Computation*, 72(243):1453–1471, 2003. doi:10.1090/s0025-5718-03-01511-4.

[LH00] Josef Leydold and Wolfgang Hörmann. Universal algorithms as an alternative for generating non-uniform continuous random variates. In *Proceedings of the International Conference on Monte Carlo Simulation 2000.*, pages 177–183, 2000.

[MT00a] George Marsaglia and Wai Wan Tsang. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):363–372, 2000. doi:10.1145/358407.358414.

[MT00b] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of statistical software*, 5(1):1–7, 2000. doi:10.18637/jss.v005.i08.

[NP09] Martina Nardon and Paolo Pianca. Simulation techniques for generalized Gaussian densities. *Journal of Statistical Computation and Simulation*, 79(11):1317–1329, 2009. doi:10.1080/00949650802290912.

[O'N14] Melissa E. O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[PHF10] Anand Patil, David Huard, and Christopher J Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4):1, 2010. doi:10.18637/jss.v035.i04.

[R C21] R Core Team. R: A language and environment for statistical computing, 2021. https://www.R-project.org/.

[Sub23] M.T. Subbotin. On the law of frequency of error. *Mat. Sbornik*, 31(2):296–301, 1923.

[Tea21] Stan Development Team. Stan modeling language users guide and reference manual, version 2.28., 2021. https://mc-stan.org.

[TL03] Günter Tirler and Josef Leydold. Automatic non-uniform random variate generation in r. In *Proceedings of DSC*, page 2, 2003.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, pages 1–12, 2020. doi:10.1038/s41592-019-0686-2.

[VN51] John Von Neumann. Various techniques used in connection with random digits. *Appl. Math Ser*, 12(36-38):3, 1951.

[Wal77] Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977. doi:10.1145/355744.355749.

Utilizing SciPy and other open source packages to provide a powerful API for materials manipulation in the Schrödinger Materials Suite

Alexandr Fonari[‡*], Farshad Fallah[‡], Michael Rauch[‡]

Abstract—The use of several open source scientific packages in the Schrödinger Materials Science Suite will be discussed. A typical workflow for materials discovery will be described, discussing how open source packages have been incorporated at every stage. Some recent implementations of machine learning for materials discovery will be discussed, as well as how open source packages were leveraged to achieve results faster and more efficiently.

Index Terms—materials, active learning, OLED, deposition, evaporation

Introduction

A common materials discovery practice or workflow is to start with reading an experimental structure of a material or generating a structure in silico, computing its properties of interest (e.g. elastic constants, electrical conductivity), tuning the material by modifying its structure (e.g. doping) or adding and removing atoms (deposition, evaporation), and then recomputing the properties of the modified material (Figure 1). Computational materials discovery leverages such workflows to empower researchers to explore vast design spaces and uncover root causes without (or in conjunction with) laboratory experimentation.

Software tools for computational materials discovery can be facilitated by utilizing existing libraries that cover the fundamental mathematics used in the calculations in an optimized fashion. This use of existing libraries allows developers to devote more time to developing new features instead of re-inventing established methods. As a result, such a complementary approach improves the performance of computational materials software and reduces overall maintenance.

The Schrödinger Materials Science Suite [LLC22] is a proprietary computational chemistry/physics platform that streamlines materials discovery workflows into a single graphical user interface (Materials Science Maestro). The interface is a single portal for structure building and enumeration, physics-based modeling and machine learning, visualization and analysis. Tying together the various modules are a wide variety of scientific packages, some of which are proprietary to Schrödinger, Inc., some of which are open-source and many of which blend the two to optimize capabilities and efficiency. For example, the main simulation engine for molecular quantum mechanics is the Jaguar [BHH+13] proprietary code. The proprietary classical molecular dynamics code Desmond (distributed by Schrödinger, Inc.) [SGB+14] is used to obtain physical properties of soft materials, surfaces and polymers. For periodic quantum mechanics, the main simulation engine is the open source code Quantum ESPRESSO (QE) [GAB+17]. One of the co-authors of this proceedings (A. Fonari) contributes to the QE code in order to make integration with the Materials Suite more seamless and less error-prone. As part of this integration, support for using the portable `XML` format for input and output in QE has been implemented in the open source Python package qeschema [BDBF].

Figure 2 gives an overview of some of the various products that compose the Schrödinger Materials Science Suite. The various workflows are implemented mainly in Python (some of them described below), calling on proprietary or open-source code where appropriate, to improve the performance of the software and reduce overall maintenance.

The materials discovery cycle can be run in a high-throughput manner, enumerating different structure modifications in a systematic fashion, such as doping ratio in a semiconductor or depositing different adsorbates. As we will detail herein, there are several open source packages that allow the user to generate a large number of structures, run calculations in high throughput manner and analyze the results. For example, the open source package pymatgen [ORJ+13] facilitates generation and analysis of periodic structures. It can generate inputs for and read outputs of QE, the commercial codes VASP and Gaussian, and several other formats. To run and manage workflow jobs in a high-throughput manner, open source packages such as Custodian [ORJ+13] and AiiDA [HZU+20] can be used.

Materials import and generation

For reading and writing of material structures, several open source packages (e.g. OpenBabel [OBJ+11], RDKit [LTK+22]) have implemented functionality for working with several commonly used formats (e.g. CIF, PDB, mol, xyz). Periodic structures of materials, mainly coming from single crystal X-ray/neutron diffraction experiments, are distributed in CIF (Crystallographic Information File), PDB (Protein Data Bank) and lately mmCIF

* *Corresponding author: sasha.fonari@schrodinger.com*
‡ *Schrödinger Inc., 1540 Broadway, 24th Floor. New York, NY 10036*

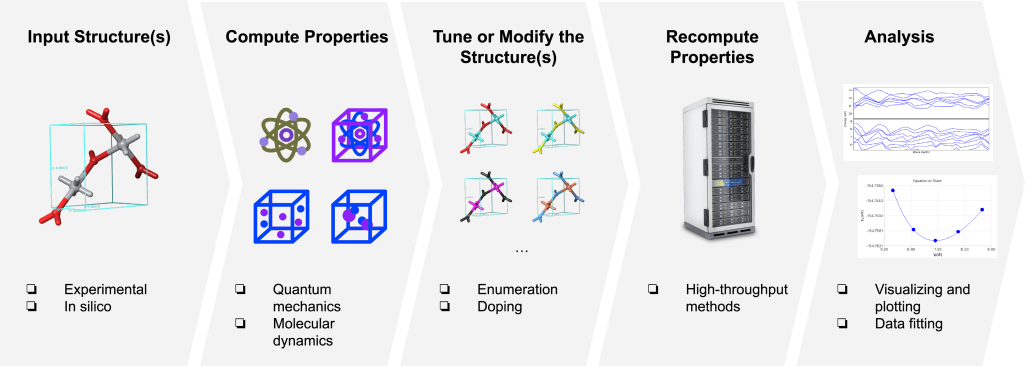


Fig. 1: *Example of a workflow for computational materials discovery.*

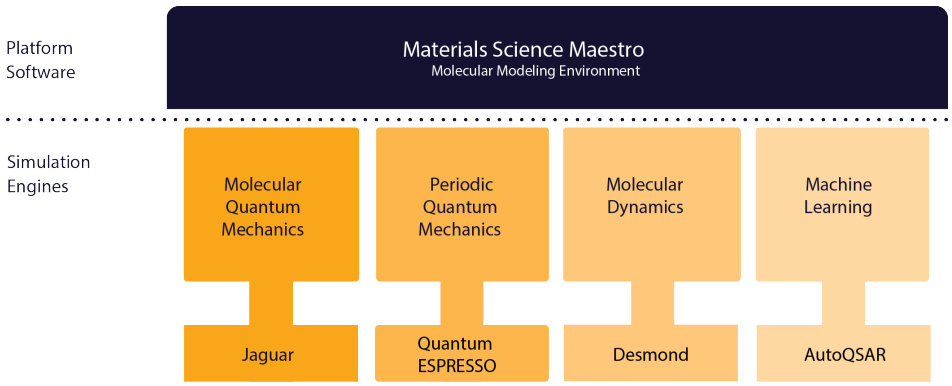


Fig. 2: *Some example products that compose the Schrödinger Materials Science Suite.*

formats [WF05]. Correctly reading experimental structures is of significant importance, since the rest of the materials discovery workflow depends on it. In addition to atom coordinates and periodic cell information, structural data also contains symmetry operations (listed explicitly or by the means of providing a space group) that can be used to decrease the number of computations required for a particular system by accounting for symmetry. This can be important, especially when scaling high-throughput calculations. From file, structure is read in a structure object through which atomic coordinates (as a NumPy array) and chemical information of the material can be accessed and updated. Structure object is similar to the one implemented in open source packages such as pymatgen [ORJ+13] and ASE [LMB+17]. All the structure manipulations during the workflows are done by using structure object interface (see structure deformation example below). Example of Structure object definition in pymatgen:

```
class Structure:

    def __init__(self, lattice, species, coords, ...):
        """Create a periodic structure."""
```

One consideration of note is that PDB, CIF and mmCIF structure formats allow description of the positional disorder (for example, a solvent molecule without a stable position within the cell which can be described by multiple sets of coordinates). Another complication is that experimental data spans an interval of almost a century: one of the oldest crystal structures deposited in the Cambridge Structural Database (CSD) [GBLW16] dates to 1924 [HM24]. These nuances and others present nontrivial technical challenges for developers. Thus, it has been a continuous effort by Schrödinger, Inc. (at least 39 commits and several weeks of work went into this project) and others to correctly read and convert periodic structures in OpenBabel. By version 3.1.1 (the most recent at writing time), the authors are not aware of any structures read incorrectly by OpenBabel. In general, non-periodic molecular formats are simpler to handle because they only contain atom coordinates but no cell or symmetry information. OpenBabel has Python bindings but due to the GPL license limitation, it is called as a subprocess from the Schrödinger Materials Suite.

Another important consideration in structure generation is modeling of substitutional disorder in solid alloys and materials with point defects (intermetallics, semiconductors, oxides and their crystalline surfaces). In such cases, the unit cell and atomic sites of the crystal or surface slab are well defined while the chemical species occupying the site may vary. In order to simulate substitutional disorder, one must generate the ensemble of structures that includes all statistically significant atomic distributions in a given unit cell. This can be achieved by a brute force enumeration of all symmetrically unique atomic structures with a given number of vacancies, impurities or solute atoms. The open source library enumlib [HF08] implements algorithms for such a systematic enumeration of periodic structures. The enumlib package consists of several Fortran binaries and Python scripts that can be run as a subprocess (no Python bindings). This allows the user to generate a large set of symmetrically nonequivalent materials with different compositions (e.g. doping or defect concentration).

Recently, we applied this approach in simultaneous study of the activity and stability of Pt based core-shell type catalysts for the oxygen reduction reaction [MGF+19]. We generated a set of stable doped Pt/transition metal/nitrogen surfaces using periodic enumeration. Using QE to perform periodic density functional

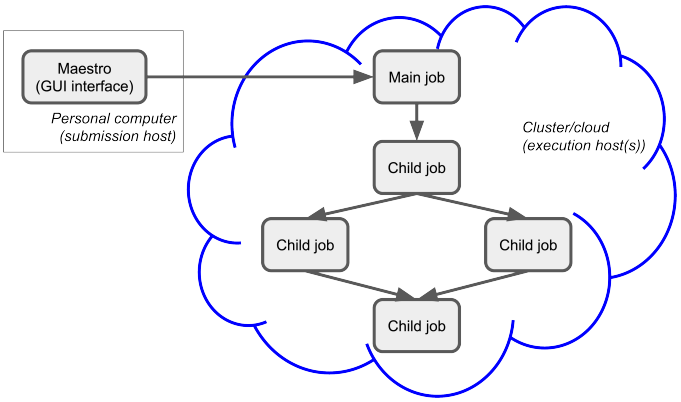


Fig. 3: *Example of the job submission process.*

theory (DFT) calculations, we assessed surface phase diagrams for Pt alloys and identified the avenues for stabilizing the cost effective core-shell systems by a judicious choice of the catalyst core material. Such catalysts may prove critical in electrocatalysis for fuel cell applications.

Workflow capabilities

In the last section, we briefly described a complete workflow from structure generation and enumeration to periodic DFT calculations to analysis. In order to be able to run a massively parallel screening of materials, a highly scalable and stable queuing system (job scheduler) is required. We have implemented a job queuing system on top of the most used queuing systems (LSF, PBS, SGE, SLURM, TORQUE, UGE) and exposed a Python API to submit and monitor jobs. In line with technological advancements, cloud is also supported by means of a virtual cluster configured with SLURM. This allows the user to submit a large number of jobs, limited only by SLURM scheduling capabilities and cloud resources. In order to accommodate job dependencies in workflows, for each job, a parent job (or multiple parent jobs) can be defined forming a directed graph of jobs (Figure 3).

There could be several reasons for a job to fail. Depending on the reason of failure, there are several restart and recovery mechanisms in place. The lowest level is the restart mechanism (in SLURM it is called `requeue`) which is performed by the queuing system itself. This is triggered when a node goes down. On the cloud, preemptible instances (nodes) can go offline at any moment. In addition, workflows implemented in the proprietary Schrödinger Materials Science Suite have built-in methods for handling various types of failure. For example, if the simulation is not converging to a requested energy accuracy, it is wasteful to blindly restart the calculation without changing some input parameters. However, in the case of a failure due to full disk space, it is reasonable to try restart with hopes to get a node with more empty disk space. If a job fails (and cannot be restarted), all its children (if any) will not start, thus saving queuing and computational time.

Having developed robust systems for running calculations, job queuing and troubleshooting (autonomously, when applicable), the developed workflows have allowed us and our customers to perform massive screenings of materials and their properties. For example, we reported a massive screening of 250,000 charge-conducting organic materials, totaling approximately 3,619,000 DFT SCF (self-consistent field) single-molecule calculations using Jaguar that took 457,265 CPU hours (~52 years) [MAS+20]. Another similar case study is the high-throughput molecular dynamics simulations (MD) of thermophysical properties of polymers for various applications [ABG+21]. There, using Desmond we computed the glass transition temperature (T_g) of 315 polymers and compared the results with experimental measurements [Bic02]. This study took advantage of GPU (graphics processing unit) support as implemented in Desmond, as well as the job scheduler API described above.

Other workflows implemented in the Schrödinger Materials Science Suite utilize open source packages as well. For soft materials (polymers, organic small molecules and substrates composed of soft molecules), convex hull and related mathematical methods are important for finding possible accessible solvent voids (during submerging or sorption) and adsorbate sites (during molecular deposition). These methods are conveniently implemented in the open source SciPy [VGO+20] and NumPy [HMvdW+20] packages. Thus, we implemented molecular deposition and evaporation workflows by using the Desmond MD engine as the backend in tandem with the convex hull functionality. This workflow enables simulation of the deposition and evaporation of the small molecules on a substrate. We utilized the aforementioned deposition workflow in the study of organic light-emitting diodes (OLEDs), which are fabricated using a stepwise process, where new layers are deposited on top of previous layers. Both vacuum and solution deposition processes have been used to prepare these films, primarily as amorphous thin film active layers lacking long-range order. Each of these deposition techniques introduces changes to the film structure and consequently, different charge-transfer and luminescent properties [WKB+22].

As can be seen from above, a workflow is usually some sort of structure modification through the structure object with a subsequent call to a backend code and analysis of its output if it succeeds. Input for the next iteration depends on the output of the previous iteration in some workflows. Due to the large chemical and manipulation space of the materials, sometimes it very tricky to keep code for all workflows follow the same code logic. For every workflow and/or functionality in the Materials Science Suite, some sort of peer reviewed material (publication, conference presentation) is created where implemented algorithms are described to facilitate reproducibility.

Data fitting algorithms and use cases

Materials simulation engines for QM, periodic DFT, and classical MD (referred to herein as backends) are frequently written in compiled languages with enabled parallelization for CPU or GPU hardware. These backends are called from Python workflows using the job queuing systems described above. Meanwhile, packages such as SciPy and NumPy provide sophisticated numerical function optimization and fitting capabilities. Here, we describe examples of how the Schrödinger suite can be used to combine materials simulations with popular optimization routines in the SciPy ecosystem.

Recently we implemented convex analysis of the stress strain curve (as described here [PKD18]). `scipy.optimize.minimize` is used for a constrained minimization with boundary conditions of a function related to the stress strain curve. The stress strain curve is obtained from a series of MD simulations on deformed cells (cell deformations are defined by strain type and deformation step). The pressure

tensor of a deformed cell is related to stress. This analysis allowed prediction of elongation at yield for high density polyethylene polymer. Figure 4 shows obtained calculated yield of 10% vs. experimental value within 9-18% range [BAS+20].

The `scipy.optimize` package is used for a least-squares fit of the bulk energies at different cell volumes (compressed and expanded) in order to obtain the bulk modulus and equation of state (EOS) of a material. In the Schrödinger suite this was implemented as a part of an EOS workflow, in which fitting is performed on the results obtained from a series of QE calculations performed on the original as well as compressed and expanded (deformed) cells. An example of deformation applied to a structure in pymatgen:

```
from pymatgen.analysis.elasticity import strain
from pymatgen.core import lattice
from pymatgen.core import structure

deform = strain.Deformation([
    [1.0, 0.02, 0.02],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]])

latt = lattice.Lattice([
    [3.84, 0.00, 0.00],
    [1.92, 3.326, 0.00],
    [0.00, -2.22, 3.14],
])

st = structure.Structure(
    latt,
    ["Si", "Si"],
    [[0, 0, 0], [0.75, 0.5, 0.75]])

strained_st = deform.apply_to_structure(st)
```

This is also an example of loosely coupled (embarrassingly parallel) jobs. In particular, calculations of the deformed cells only depend on the bulk calculation and do not depend on each other. Thus, all the deformation jobs can be submitted in parallel, facilitating high-throughput runs.

Structure refinement from powder diffraction experiment is another example where more complex optimization is used. Powder diffraction is a widely used method in drug discovery to assess purity of the material and discover known or unknown crystal polymorphs [KBD+21]. In particular, there is interest in fitting of the experimental powder diffraction intensity peaks to the indexed peaks (Pawley refinement) [JPS92]. Here we employed the open source lmfit package [NSA+16] to perform a minimization of the multivariable Voigt-like function that represents the entire diffraction spectrum. This allows the user to refine (optimize) unit cell parameters coming from the indexing data and as the result, goodness of fit (*R*-factor) between experimental and simulated spectrum is minimized.

Machine learning techniques

Of late, there is great interest in machine learning assisted materials discovery. There are several components required to perform machine learning assisted materials discovery. In order to train a model, benchmark data from simulation and/or experimental data is required. Besides benchmark data, computation of the relevant descriptors is required (see below). Finally, a model based on benchmark data and descriptors is generated that allows prediction of properties for novel materials. There are several techniques to generate the model, such as linear or non-linear fitting to neural networks. Tools include the open source DeepChem [REW+19] and AutoQSAR [DDS+16] from the Schrödinger suite. Depending on the type of materials, benchmark data can be obtained using different codes available in the Schrödinger suite:

- small molecules and finite systems - Jaguar
- periodic systems - Quantum ESPRESSO
- larger polymeric and similar systems - Desmond

Different materials systems require different descriptors for featurization. For example, for crystalline periodic systems, we have implemented several sets of tailored descriptors. Generation of these descriptors again uses a mix of open source and Schrödinger proprietary tools. Specifically:

- elemental features such as atomic weight, number of valence electrons in *s*, *p* and *d*-shells, and electronegativity
- structural features such as density, volume per atom, and packing fraction descriptors implemented in the open source matminer package [WDF+18]
- intercalation descriptors such as cation and anion counts, crystal packing fraction, and average neighbor ionicity [SYC+17] implemented in the Schrödinger suite
- three-dimensional smooth overlap of atomic positions (SOAP) descriptors implemented in the open source DScribe package [HJM+20].

We are currently training models that use these descriptors to predict properties, such as bulk modulus, of a set of Li-containing battery related compounds [Cha]. Several models will be compared, such as kernel regression methods (as implemented in the open source scikit-learn code [PVG+11]) and AutoQSAR.

For isolated small molecules and extended non-periodic systems, RDKit can be used to generate a large number of atomic and molecular descriptors. A lot of effort has been devoted to ensure that RDKit can be used on a wide variety of materials that are supported by the Schrödinger suite. At the time of writing, the 4th most active contributor to RDKit is Ricardo Rodriguez-Schmidt from Schrödinger [RDK].

Recently, active learning (AL) combined with DFT has received much attention to address the challenge of leveraging exhaustive libraries in materials informatics [VPB21], [SPA+19]. On our side, we have implemented a workflow that employs active learning (AL) for intelligent and iterative identification of promising materials candidates within a large dataset. In the framework of AL, the predicted value with associated uncertainty is considered to decide what materials to be added in each iteration, aiming to improve the model performance in the next iteration (Figure 5).

Since it could be important to consider multiple properties simultaneously in material discovery, multiple property optimization (MPO) has also been implemented as a part of the AL workflow [KAG+22]. MPO allows scaling and combining multiple properties into a single score. We employed the AL workflow to determine the top candidates for hole (positively charged carrier) transport layer (HTL) by evaluating 550 molecules in 10 iterations using DFT calculations for a dataset of ~9,000 molecules [AKA+22]. Resulting model was validated by randomly picking a molecule from the dataset, computing properties with DFT and comparing those to the predicted values. According to the semi-classical Marcus equation [Mar93], high rates of hole transfer are inversely proportional to hole reorganization energies. Thus, MPO scores were computed based on minimizing hole reorganization energy and targeting oxidation potential to an appropriate level to ensure a low energy barrier for hole injection from the anode

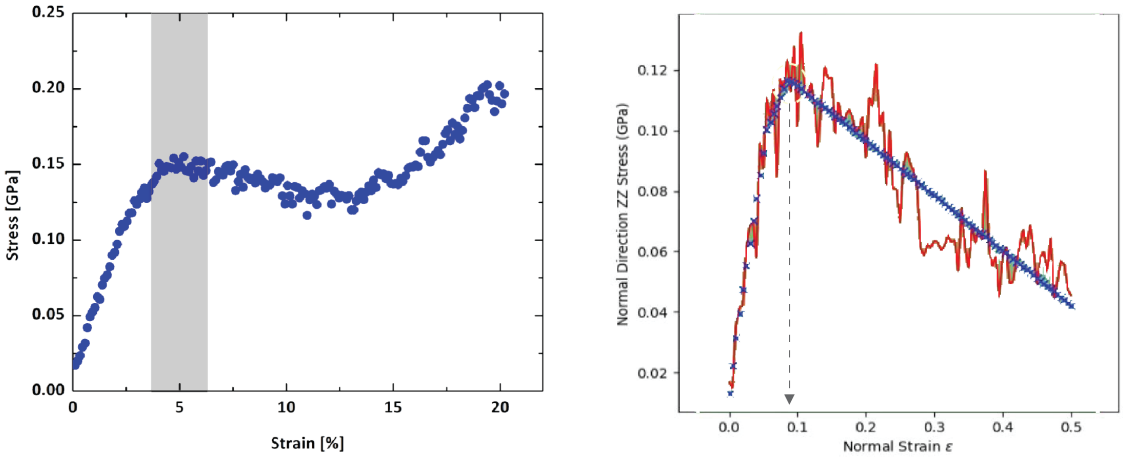


Fig. 4: *Left: The uniaxial stress/strain curve of a polymer calculated using Desmond through the stress strain workflow. The dark grey band indicates an inflection that marks the yield point. Right: Constant strain simulation with convex analysis indicates elongation at yield. The red curve shows simulated stress versus strain. The blue curve shows convex analysis.*

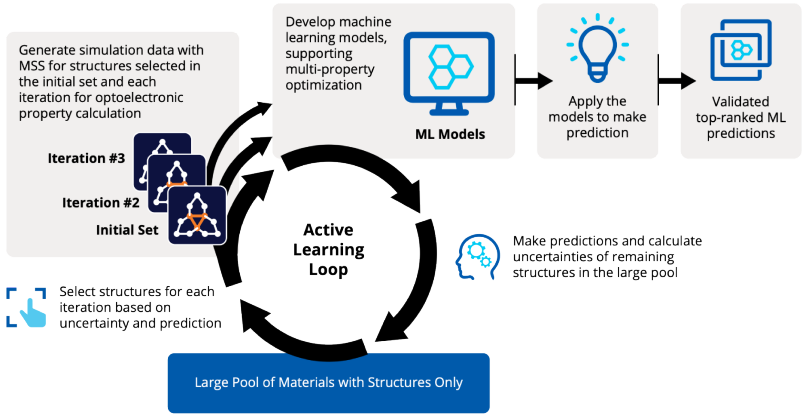


Fig. 5: *Active learning workflow for the design and discovery of novel optoelectronics molecules.*

into the emissive layer. In this workflow, we used RDKit to compute descriptors for the chemical structures. These descriptors generated on the initial subset of structures are given as vectors to an algorithm based on Random Forest Regressor as implemented in scikit-learn. Bayesian optimization is employed to tune the hyperparameters of the model. In each iteration, a trained model is applied for making predictions on the remaining materials in the dataset. Figure 6 (A) displays MPO scores for the HTL dataset estimated by AL as a function of hole reorganization energies that are separately calculated for all the materials. This figure indicates that there are many materials in the dataset with desired low hole reorganization energies but are not suitable for HTL due to their improper oxidation potentials, suggesting that MPO is important to evaluate the optoelectronic performance of the materials. Figure 6 (B) presents MPO scores of the materials used in the training dataset of AL, demonstrating that the feedback loop in the AL workflow efficiently guides the data collection as the size of the training set increases.

To appreciate the computational efficiency of such an approach, it is worth noting that performing DFT calculations for all of the 9,000 molecules in the dataset would increase the computational cost by a factor of 15 versus the AL workflow. It seems that AL approach can be useful in the cases where problem space is broad (like chemical space), but there are many clusters of similar items (similar molecules). In this case, benchmark data is only needed for few representatives of each cluster. We are currently working on applying this approach to train models for predicting physical properties of soft materials (polymers).

Conclusions

We present several examples of how Schrödinger Materials Suite integrates open source software packages. There is a wide range of applications in materials science that can benefit from already existing open source code. Where possible, we report issues to the package authors and submit improvements and bug fixes in the form of the pull requests. We are thankful to all who have contributed to open source libraries, and have made it possible for us to develop a platform for accelerating innovation in materials and drug discovery. We will continue contributing to these projects and we hope to further give back to the scientific community by facilitating research in both academia and industry. We hope that this report will inspire other scientific companies to give back to the open source community in order to improve the computational materials field and make science more reproducible.

Acknowledgments

The authors acknowledge Bradley Dice and Wenduo Zhou for their valuable comments during the review of the manuscript.

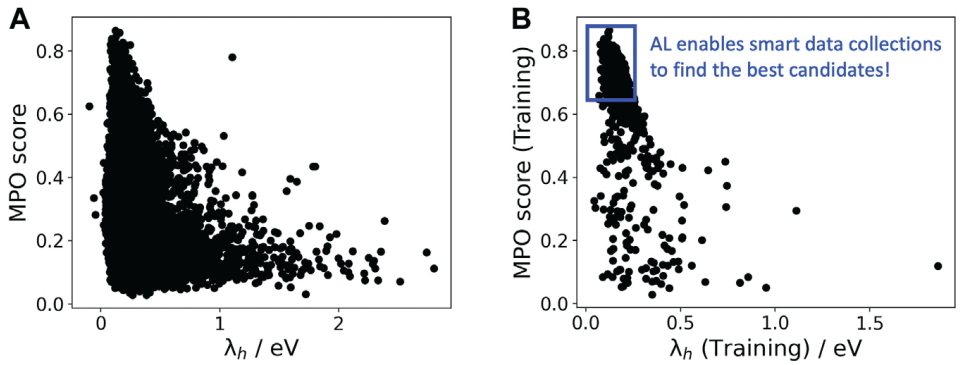


Fig. 6: *A: MPO score of all materials in the HTL dataset. B: Those used in the training set as a function of the hole reorganization energy (λ_h).*

REFERENCES

[ABG+21] Mohammad Atif Faiz Afzal, Andrea R. Browning, Alexander Goldberg, Mathew D. Halls, Jacob L. Gavartin, Tsuguo Morisato, Thomas F. Hughes, David J. Giesen, and Joseph E. Goose. High-throughput molecular dynamics simulations and validation of thermophysical properties of polymers for various applications. *ACS Applied Polymer Materials*, 3, 2021. doi:10.1021/acsapm.0c00524.

[AKA+22] Hadi Abroshan, H. Shaun Kwak, Yuling An, Christopher Brown, Anand Chandrasekaran, Paul Winget, and Mathew D. Halls. Active learning accelerates design and optimization of hole-transporting materials for organic electronics. *Frontiers in Chemistry*, 9, 2022. doi:10.3389/fchem.2021.800371.

[BAS+20] A. R. Browning, M. A. F. Afzal, J. Sanders, A. Goldberg, A. Chandrasekaran, and H. S. Kwak. Polyolefin molecular simulation for critical physical characteristics. *International Polyolefins Conference*, 2020.

[BDBF] Davide Brunato, Pietro Delugas, Giovanni Borghi, and Alexandr Fonari. qeschema. URL: https://github.com/QEF/qeschema.

[BHH+13] Art D. Bochevarov, Edward Harder, Thomas F. Hughes, Jeremy R. Greenwood, Dale A. Braden, Dean M. Philipp, David Rinaldo, Mathew D. Halls, Jing Zhang, and Richard A. Friesner. Jaguar: A high-performance quantum chemistry software program with strengths in life and materials sciences. *International Journal of Quantum Chemistry*, 113, 2013. doi:10.1002/qua.24481.

[Bic02] Jozef Bicerano. *Prediction of polymer properties*. cRc Press, 2002.

[Cha] A. Chandrasekaran. Active learning accelerated design of ionic materials. *in progress*.

[DDS+16] Steven L. Dixon, Jianxin Duan, Ethan Smith, Christopher D. Von Bargen, Woody Sherman, and Matthew P. Repasky. Autoqsar: An automated machine learning tool for best-practice quantitative structure-activity relationship modeling. *Future Medicinal Chemistry*, 8, 2016. doi:10.4155/fmc-2016-0093.

[GAB+17] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. Buongiorno Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. Dal Corso, S. De Gironcoli, P. Delugas, R. A. Distasio, A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, J. Jia, M. Kawamura, H. Y. Ko, A. Kokalj, E. Kücükbenli, M. Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H. V. Nguyen, A. Otero-De-La-Roza, L. Paulatto, S. Poncé, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu, and S. Baroni. Advanced capabilities for materials modelling with quantum espresso. *Journal of Physics Condensed Matter*, 29, 2017. URL: https://www.quantum-espresso.org/, doi:10.1088/1361-648X/aa8f79.

[GBLW16] Colin R. Groom, Ian J. Bruno, Matthew P. Lightfoot, and Suzanna C. Ward. The cambridge structural database. *Acta Crystallographica Section B: Structural Science, Crystal Engineering and Materials*, 72, 2016. doi:10.1107/S2052520616003954.

[HF08] Gus L.W. Hart and Rodney W. Forcade. Algorithm for generating derivative structures. *Physical Review B - Condensed Matter and Materials Physics*, 77, 2008. URL: https://github.com/msg-byu/enumlib/, doi:10.1103/PhysRevB.77.224115.

[HJM+20] Lauri Himanen, Marc O.J. Jager, Eiaki V. Morooka, Filippo Federici Canova, Yashasvi S. Ranawat, David Z. Gao, Patrick Rinke, and Adam S. Foster. Dscribe: Library of descriptors for machine learning in materials science. *Computer Physics Communications*, 247, 2020. URL: https://singroup.github.io/dscribe/latest/, doi:10.1016/j.cpc.2019.106949.

[HM24] O Hassel and H Mark. The crystal structure of graphite. *Physik. Z*, 25:317–337, 1924.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy, 2020. URL: https://numpy.org/, doi:10.1038/s41586-020-2649-2.

[HZU+20] Sebastiaan P. Huber, Spyros Zoupanos, Martin Uhrin, Leopold Talirz, Leonid Kahle, Rico Hauselmann, Dominik Gresch, Tiziano Müller, Aliaksandr V. Yakutovich, Casper W. Andersen, Francisco F. Ramirez, Carl S. Adorf, Fernando Gargiulo, Snehal Kumbhar, Elsa Passaro, Conrad Johnston, Andrius Merkys, Andrea Cepellotti, Nicolas Mounet, Nicola Marzari, Boris Kozinsky, and Giovanni Pizzi. Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Scientific Data*, 7, 2020. URL: https://www.aiida.net/, doi:10.1038/s41597-020-00638-4.

[JPS92] J. Jansen, R. Peschar, and H. Schenk. Determination of accurate intensities from powder diffraction data. i. whole-pattern fitting with a least-squares procedure. *Journal of Applied Crystallography*, 25, 1992. doi:10.1107/S0021889891012104.

[KAG+22] H. Shaun Kwak, Yuling An, David J. Giesen, Thomas F. Hughes, Christopher T. Brown, Karl Leswing, Hadi Abroshan, and Mathew D. Halls. Design of organic electronic materials with a goal-directed generative model powered by deep neural networks and high-throughput molecular simulations. *Frontiers in Chemistry*, 9, 2022. doi:10.3389/fchem.2021.800370.

[KBD+21] James A Kaduk, Simon J L Billinge, Robert E Dinnebier, Nathan Henderson, Ian Madsen, Radovan Černý, Matteo Leoni, Luca Lutterotti, Seema Thakral, and Daniel Chateigner. Powder diffraction. *Nature Reviews Methods Primers*, 1:77, 2021. URL: https://doi.org/10.1038/s43586-021-00074-7, doi:10.1038/s43586-021-00074-7.

[LLC22] Schrödinger LLC. Schrödinger release 2022-2: Materials science suite, 2022. URL: https://www.schrodinger.com/platform/materials-science.

[LMB+17] Ask Hjorth Larsen, Jens JØrgen Mortensen, Jakob Blomqvist, Ivano E. Castelli, Rune Christensen, Marcin Dułak, Jesper Friis, Michael N. Groves, BjØrk Hammer, Cory Hargus, Eric D. Hermes, Paul C. Jennings, Peter Bjerre Jensen, James Kermode, John R. Kitchin, Esben Leonhard Kolsbjerg, Joseph Kubal, Kristen Kaasbjerg, Steen Lysgaard, Jón Bergmann Maronsson, Tristan Maxson, Thomas Olsen, Lars Pastewka, Andrew Peterson, Carsten Rostgaard, Jakob SchiØtz, Ole Schütt, Mikkel Strange, Kristian S. Thygesen, Tejs Vegge, Lasse Vilhelmsen, Michael Walter, Zhenhua Zeng, and Karsten W. Jacobsen. The atomic simulation environment - a python library for working with atoms, 2017. URL: https://wiki.fysik.dtu.dk/ase/, doi:10.1088/1361-648X/aa680e.

[LTK+22] Greg Landrum, Paolo Tosco, Brian Kelley, Ric, sriniker, gedeck, Riccardo Vianello, NadineSchneider, Eisuke Kawashima, Andrew Dalke, Dan N, David Cosgrove, Gareth Jones, Brian Cole, Matt Swain, Samo Turk, AlexanderSavelyev, Alain Vaucher, Maciej Wójcikowski, Ichiru Take, Daniel Probst, Kazuya Ujihara, Vincent F. Scalfani, guillaume godin, Axel Pahl, Francois Berenger, JLVarjo, strets123, JP, and DoliathGavid. rdkit. 6 2022. URL: https://rdkit.org/, doi:10.5281/ZENODO.6605135.

[Mar93] Rudolph A. Marcus. Electron transfer reactions in chemistry. theory and experiment. *Reviews of Modern Physics*, 65, 1993. doi:10.1103/RevModPhys.65.599.

[MAS+20] Nobuyuki N. Matsuzawa, Hideyuki Arai, Masaru Sasago, Eiji Fujii, Alexander Goldberg, Thomas J. Mustard, H. Shaun Kwak, David J. Giesen, Fabio Ranalli, and Mathew D. Halls. Massive theoretical screen of hole conducting organic materials in the heteroacene family by using a cloud-computing environment. *Journal of Physical Chemistry A*, 124, 2020. doi:10.1021/acs.jpca.9b10998.

[MGF+19] Thomas Mustard, Jacob Gavartin, Alexandr Fonari, Caroline Krauter, Alexander Goldberg, H Kwak, Tsuguo Morisato, Sudharsan Pandiyan, and Mathew Halls. Surface reactivity and stability of core-shell solid catalysts from ab initio combinatorial calculations. volume 258, 2019.

[NSA+16] Matthew Newville, Till Stensitzki, Daniel B Allen, Michal Rawlik, Antonino Ingargiola, and Andrew Nelson. Lmfit: Non-linear least-square minimization and curve-fitting for python. *Astrophysics Source Code Library*, page ascl–1606, 2016. URL: https://lmfit.github.io/lmfit-py/.

[OBJ+11] Noel M. O'Boyle, Michael Banck, Craig A. James, Chris Morley, Tim Vandermeersch, and Geoffrey R. Hutchison. Open babel: An open chemical toolbox. *Journal of Cheminformatics*, 3, 2011. URL: https://openbabel.org/, doi:10.1186/1758-2946-3-33.

[ORJ+13] Shyue Ping Ong, William Davidson Richards, Anubhav Jain, Geoffroy Hautier, Michael Kocher, Shreyas Cholia, Dan Gunter, Vincent L. Chevrier, Kristin A. Persson, and Gerbrand Ceder. Python materials genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, 68, 2013. URL: https://pymatgen.org/, doi:10.1016/j.commatsci.2012.10.028.

[PKD18] Paul N. Patrone, Anthony J. Kearsley, and Andrew M. Dienstfrey. The role of data analysis in uncertainty quantification: Case studies for materials modeling. volume 0, 2018. doi:10.2514/6.2018-0927.

[PVG+11] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12, 2011. URL: https://scikit-learn.org/.

[RDK] Rdkit contributors. URL: https://github.com/rdkit/rdkit/graphs/contributors.

[REW+19] Bharath Ramsundar, Peter Eastman, Patrick Walters, Vijay Pande, Karl Leswing, and Zhenqin Wu. *Deep Learning for the Life Sciences*. O'Reilly Media, 2019. https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.

[SGB+14] David E. Shaw, J. P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk Hai Mok, Mark A. Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. volume 2015-January, 2014. doi:10.1109/SC.2014.9.

[SPA+19] Gabriel R. Schleder, Antonio C.M. Padilha, Carlos Mera Acosta, Marcio Costa, and Adalberto Fazzio. From dft to machine learning: Recent approaches to materials science - a review. *JPhys Materials*, 2, 2019. doi:10.1088/2515-7639/ab084b.

[SYC+17] Austin D Sendek, Qian Yang, Ekin D Cubuk, Karel-Alexander N Duerloo, Yi Cui, and Evan J Reed. Holistic computational structure screening of more than 12000 candidates for solid lithium-ion conductor materials. *Energy and Environmental Science*, 10:306–320, 2017. doi:10.1039/c6ee02697d.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R.J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Haggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T. Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vinícius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, and Yoshiki Vázquez-Baeza. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17, 2020. doi:10.1038/s41592-019-0686-2.

[VPB21] Rama Vasudevan, Ghanshyam Pilania, and Prasanna V. Balachandran. Machine learning for materials design and discovery. *Journal of Applied Physics*, 129, 2021. doi:10.1063/5.0043300.

[WDF+18] Logan Ward, Alexander Dunn, Alireza Faghaninia, Nils E.R. Zimmermann, Saurabh Bajaj, Qi Wang, Joseph Montoya, Jiming Chen, Kyle Bystrom, Maxwell Dylla, Kyle Chard, Mark Asta, Kristin A. Persson, G. Jeffrey Snyder, Ian Foster, and Anubhav Jain. Matminer: An open source toolkit for materials data mining. *Computational Materials Science*, 152, 2018. URL: https://hackingmaterials.lbl.gov/matminer/, doi:10.1016/j.commatsci.2018.05.018.

[WF05] John D. Westbrook and Paula M.D. Fitzgerald. The pdb format, mmcif formats, and other data formats, 2005. doi:10.1002/0471721204.ch8.

[WKB+22] Paul Winget, H. Shaun Kwak, Christopher T. Brown, Alexandr Fonari, Kevin Tran, Alexander Goldberg, Andrea R. Browning, and Mathew D. Halls. Organic thin films for oled applications: Influence of molecular structure, deposition method,

and deposition conditions. *International Conference on the Science and Technology of Synthetic Metals*, 2022.

A Novel Pipeline for Cell Instance Segmentation, Tracking and Motility Classification of Toxoplasma Gondii in 3D Space

Seyed Alireza Vaezi[‡*], Gianni Orlando[‡], Mojtaba Fazli[§], Gary Ward[¶], Silvia Moreno[‡], Shannon Quinn[‡]

Abstract—Toxoplasma gondii is the parasitic protozoan that causes disseminated toxoplasmosis, a disease that is estimated to infect around one-third of the world's population. While the disease is commonly asymptomatic, the success of the parasite is in large part due to its ability to easily spread through nucleated cells. The virulence of T. gondii is predicated on the parasite's motility. Thus the inspection of motility patterns during its lytic cycle has become a topic of keen interest. Current cell tracking projects usually focus on cell images captured in 2D which are not a true representation of the actual motion of a cell. Current 3D tracking projects lack a comprehensive pipeline covering all phases of preprocessing, cell detection, cell instance segmentation, tracking, and motion classification, and merely implement a subset of the phases. Moreover, current 3D segmentation and tracking pipelines are not targeted for users with less experience in deep learning packages. Our pipeline, *TSeg*, on the other hand, is developed for segmenting, tracking, and classifying the motility phenotypes of T. gondii in 3D microscopic images. Although TSeg is built initially focusing on T. gondii, it provides generic functions to allow users with similar but distinct applications to use it off-the-shelf. Interacting with all of TSeg's modules is possible through our Napari plugin which is developed mainly off the familiar SciPy scientific stack. Additionally, our plugin is designed with a user-friendly GUI in Napari which adds several benefits to each step of the pipeline such as visualization and representation in 3D. TSeg proves to fulfill a better generalization, making it capable of delivering accurate results with images of other cell types.

Introduction

Quantitative cell research often requires the measurement of different cell properties including size, shape, and motility. This step is facilitated using segmentation of imaged cells. With fluorescent markers, computational tools can be used to complete segmentation and identify cell features and positions over time. 2D measurements of cells can be useful, but the more difficult task of deriving 3D information from cell images is vital for metrics such as motility and volumetric qualities.

Toxoplasmosis is an infection caused by the intracellular parasite Toxoplasma gondii. T. gondii is one of the most successful parasites, infecting at least one-third of the world's population. Although Toxoplasmosis is generally benign in healthy individuals, the infection has fatal implications in fetuses and immunocompromised individuals [SG12] . T. gondii's virulence is directly linked to its lytic cycle which is comprised of invasion, replication, egress, and motility. Studying the motility of T. gondii is crucial in understanding its lytic cycle in order to develop potential treatments.

For this reason, we present a novel pipeline to detect, segment, track, and classify the motility pattern of T. gondii in 3D space. One of the main goals is to make our pipeline intuitively easy to use so that the users who are not experienced in the fields of machine learning (ML), deep learning (DL), or computer vision (CV) can still benefit from it. The other objective is to equip it with the most robust and accurate set of segmentation and detection tools so that the end product has a broad generalization, allowing it to perform well and accurately for various cell types right off the shelf.

PlantSeg uses a variant of 3D U-Net, called Residual 3D U-Net, for preprocessing and segmentation of multiple cell types [WCV+20]. PlantSeg performs best among Deep Learning algorithms for 3D Instance Segmentation and is very robust against image noise [KPR+21]. The segmentation module also includes the optional use of CellPose [SWMP21]. CellPose is a generalized segmentation algorithm trained on a wide range of cell types and is the first step toward increased optionality in TSeg. The Cell Tracking module consolidates the cell particles across the z-axis to materialize cells in 3D space and estimates centroids for each cell. The tracking module is also responsible for extracting the trajectories of cells based on the movements of centroids throughout consecutive video frames, which is eventually the input of the motion classifier module.

Most of the state-of-the-art pipelines are restricted to 2D space which is not a true representative of the actual motion of the organism. Many of them require knowledge and expertise in programming, or in machine learning and deep learning models and frameworks, thus limiting the demographic of users that can use them. All of them solely include a subset of the aforementioned modules (i.e. detection, segmentation, tracking, and classification) [SWMP21]. Many pipelines rely on the user to train their own model, hand-tailored for their specific application. This demands high levels of experience and skill in ML/DL and consequently undermines the possibility and feasibility of quickly utilizing an off-the-shelf pipeline and still getting good results.

To address these we present TSeg. It segments T. gondii cells

* *Corresponding author: sv22900@uga.edu*
‡ *University of Georgia*
§ *harvard University*
¶ *University of Vermont*

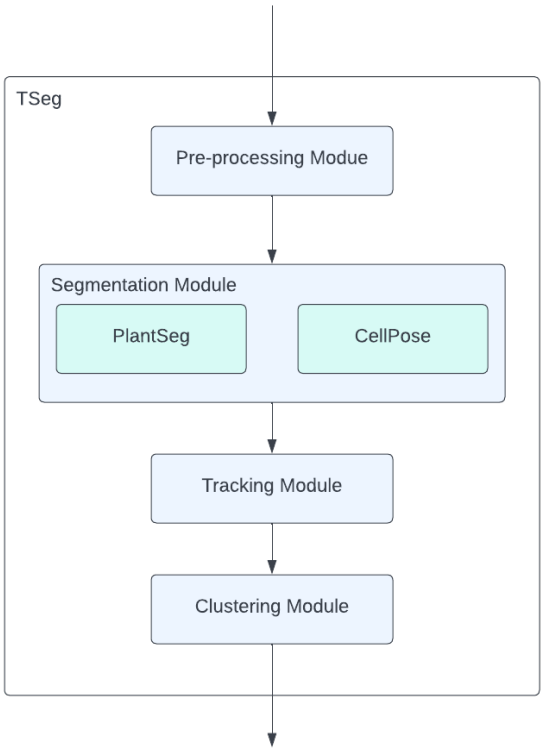


Fig. 1: *The overview of TSeg's architecture.*

in 3D microscopic images, tracks their trajectories, and classifies the motion patterns observed throughout the 3D frames. TSeg is comprised of four modules: pre-processing, segmentation, tracking, and classification. We developed TSeg as a plugin for Napari [SLE+22] - an open-source fast and interactive image viewer for Python designed for browsing, annotating, and analyzing large multi-dimensional images. Having TSeg implemented as a part of Napari not only provides a user-friendly design but also gives more advanced users the possibility to attach and execute their custom code and even interact with the steps of the pipeline if needed. The preprocessing module is equipped with basic and extra filters and functionalities to aid in the preparation of the input data. TSeg gives its users the advantage of utilizing the functionalities that PlantSeg and CellPose provide. These functionalities can be chosen in the pre-processing, detection, and segmentation steps. This brings forth a huge variety of algorithms and pre-built models to select from, making TSeg not only a great fit for T. gindii, but also a variety of different cell types.

The rest of this paper is structured as follows: After briefly reviewing the literature in Related Work, we move on to thoroughly describe the details of our work in the Method section. Following that, the Results section depicts the results of comprehensive tests of our plugin on T. gondii cells.

Related Work

The recent solutions in generalized and automated segmentation tools are focused on 2D cell images. Segmentation of cellular structures in 2D is important but not representative of realistic environments. Microbiological organisms are free to move on the z-axis and tracking without taking this factor into account cannot guarantee a full representation of the actual motility patterns. As an example, Fazli et al. [FVMQ18] identified three distinct motility types for T. gondii with two-dimensional data, however, they also acknowledge and state that based established heuristics from previous works there are more than three motility phenotypes for T. gondii. The focus on 2D research is understandable due to several factors. 3D data is difficult to capture as tools for capturing 3D slices and the computational requirements for analyzing this data are not available in most research labs. Most segmentation tools are unable to track objects in 3D space as the assignment of related centroids is more difficult. The additional noise from capture and focus increases the probability of incorrect assignment. 3D data also has issues with overlapping features and increased computation required per frame of time.

Fazli et al. [FVMQ18] studies the motility patterns of T. gondii and provides a computational pipeline for identifying motility phenotypes of T. gondii in an unsupervised, data-driven way. In that work Ca2+ is added to T. gondii cells inside a Fetal Bovine Serum. T. gondii cells react to Ca2+ and become motile and fluorescent. The images of motile T. gondii cells were captured using an LSM 710 confocal microscope. They use Python 3 and associated scientific computing libraries (NumPy, SciPy, scikit-learn, matplotlib) in their pipeline to track and cluster the trajectories of T. gondii. Based on this work Fazli et al. [FVM+18] work on another pipeline consisting of preprocessing, sparsification, cell detection, and cell tracking modules to track T. gondii in 3D video microscopy where each frame of the video consists of image slices taken 1 micro-meters of focal depth apart along the z-axis direction. In their latest work Fazli et al. [FSA+19] developed a lightweight and scalable pipeline using task distribution and parallelism. Their pipeline consists of multiple modules: reprocessing, sparsification, cell detection, cell tracking, trajectories extraction, parametrization of the trajectories, and clustering. They could classify three distinct motion patterns in T. gondii using the same data from their previous work.

While combining open source tools is not a novel architecture, little has been done to integrate 3D cell tracking tools. Fazeli et al. [FRF+20] motivated by the same interest in providing better tools to non-software professionals created a 2D cell tracking pipeline. This pipeline combines Stardist [WSH+20] and TrackMate [TPS+17] for automated cell tracking. This pipeline begins with the user loading cell images and centroid approximations to the ZeroCostDL4Mic [vCLJ+21] platform. ZeroCostDL4Mic is a deep learning training tool for those with no coding expertise. Once the platform is trained and masks for the training set are made for hand-drawn annotations, the training set can be input to Stardist. Stardist performs automated object detection using Euclidean distance to probabilistically determine cell pixels versus background pixels. Lastly, Trackmate uses segmentation images to track labels between timeframes and display analytics.

This Stardist pipeline is similar in concept to TSeg. Both create an automated segmentation and tracking pipeline but TSeg is oriented to 3D data. Cells move in 3-dimensional space that is not represented in a flat plane. TSeg also does not require the manual training necessary for the other pipeline. Individuals with low technical expertise should not be expected to create masks for training or even understand the training of deep neural networks. Lastly, this pipeline does not account for imperfect datasets without the need for preprocessing. All implemented algorithms in TSeg account for microscopy images with some amount of noise.

Wen et al. [WMV+21] combines multiple existing new tech-

nologies including deep learning and presents 3DeeCellTracker. 3DeeCellTracker segments and tracks cells on 3D time-lapse images. Using a small subset of their dataset they train the deep learning architecture 3D U-Net for segmentation. For tracking, a combination of two strategies was used to increase accuracy: local cell region strategies, and spatial pattern strategy. Kapoor et al. [KC21] presents VollSeg that uses deep learning methods to segment, track, and analyze cells in 3D with irregular shape and intensity distribution. It is a Jupyter Notebook-based Python package and also has a UI in Napari. For tracking, a custom tracking code is developed based on Trackmate.

Many segmentation tools require some amount of knowledge in Machine or Deep Learning concepts. Training the neural network in creating masks is a common step for open-source segmentation tools. Automating this process makes the pipeline more accessible to microbiology researchers.

Method

Data

Our dataset consists of 11 videos of T. gondii cells under a microscope, obtained from different experiments with different numbers of cells. The videos are on average around 63 frames in length. Each frame has a stack of 41 image slices of size 500×502 pixels along the z-axis (z-slices). The z-slices are captured 1µm apart in optical focal length making them 402µm×401µm×40µm in volume. The slices were recorded in raw format as RGB TIF images but are converted to grayscale for our purpose. This data is captured using a PlanApo 20x objective (NA = 0:75) on a preheated Nikon Eclipse TE300 epifluorescence microscope. The image stacks were captured using an iXon 885 EMCCD camera (Andor Technology, Belfast, Ireland) cooled to -70oC and driven by NIS Elements software (Nikon Instruments, Melville, NY) as part of related research by Ward et al. [LRK+14]. The camera was set to frame transfer sensor mode, with a vertical pixel shift speed of 1:0 µs, vertical clock voltage amplitude of +1, readout speed of 35MHz, conversion gain of 3:8×, EM gain setting of 3 and 22 binning, and the z-slices were imaged with an exposure time of 16ms.

Software

Napari Plugin: TSeg is developed as a plugin for Napari - a fast and interactive multi-dimensional image viewer for python that allows volumetric viewing of 3D images [SLE+22]. Plugins enable developers to customize and extend the functionality of Napari. For every module of TSeg, we developed its corresponding widget in the GUI, plus a widget for file management. The widgets have self-explanatory interface elements with tooltips to guide the inexperienced user to traverse through the pipeline with ease. Layers in Napari are the basic viewable objects that can be shown in the Napari viewer. Seven different layer types are supported in Napari: *Image, Labels, Points, Shapes, Surface, Tracks,* and *Vectors*, each of which corresponds to a different data type, visualization, and interactivity [SLE+22]. After its execution, the viewable output of each widget gets added to the layers. This allows the user to evaluate and modify the parameters of the widget to get the best results before continuing to the next widget. Napari supports bidirectional communication between the viewer and the Python kernel and has a built-in console that allows users to control all the features of the viewer programmatically. This adds more flexibility and customizability to TSeg for the advanced user. The full code of TSeg is available on GitHub under the MIT open source license at https://github.com/salirezav/tseg. TSeg can be installed through Napari's plugins menu.

Computational Pipeline

Pre-Processing: Due to the fast imaging speed in data acquisition, the image slices will inherently have a vignetting artifact, meaning that the corners of the images will be slightly darker than the center of the image. To eliminate this artifact we added adaptive thresholding and logarithmic correction to the pre-processing module. Furthermore, another prevalent artifact on our dataset images was a Film-Grain noise (AKA salt and pepper noise). To remove or reduce such noise a simple gaussian blur filter and a sharpening filter are included.

Cell Detection and Segmentation: TSeg's Detection and Segmentation modules are in fact backed by PlantSeg and Cell-Pose. The Detection Module is built only based on PlantSeg's CNN Detection Module [WCV+20] , and for the Segmentation Module, only one of the three tools can be selected to be executed as the segmentation tool in the pipeline. Naturally, each of the tools demands specific interface elements different from the others since each accepts different input values and various parameters. TSeg orchestrates this and makes sure the arguments and parameters are passed to the corresponding selected segmentation tool properly and the execution will be handled accordingly. The parameters include but are not limited to input data location, output directory, and desired segmentation algorithm. This allows the end-user complete control over the process and feedback from each step of the process. The preprocessed images and relevant parameters are sent to a modular segmentation controller script. As an effort to allow future development on TSeg, the segmentation controller script shows how the pipeline integrates two completely different segmentation packages. While both PlantSeg and CellPose use conda environments, PlantSeg requires modification of a YAML file for initialization while CellPose initializes directly from command line parameters. In order to implement PlantSeg, TSeg generates a YAML file based on GUI input elements. After parameters are aligned, the conda environment for the chosen segmentation algorithm is opened in a subprocess. The *$CONDA_PREFIX* environment variable allows the bash command to start conda and context switch to the correct segmentation environment.

Tracking: Features in each segmented image are found using the scipy label function. In order to reduce any leftover noise, any features under a minimum size are filtered out and considered leftover noise. After feature extraction, centroids are calculated using the center of mass function in scipy. The centroid of the 3D cell can be used as a representation of the entire body during tracking. The tracking algorithm goes through each captured time instance and connects centroids to the likely next movement of the cell. Tracking involves a series of measures in order to avoid incorrect assignments. An incorrect assignment could lead to inaccurate result sets and unrealistic motility patterns. If the same number of features in each frame of time could be guaranteed from segmentation, minimum distance could assign features rather accurately. Since this is not a guarantee, the Hungarian algorithm must be used to associate a COST with the assignment of feature tracking. The Hungarian method is a combinatorial optimization algorithm that solves the assignment problem in polynomial time. COST for the tracking algorithm determines which feature is the next iteration of the cell's tracking through the complete time series. The combination of distance between centroids for all

previous points and the distance to the potential new centroid. If an optimal next centroid can't be found within an acceptable distance of the current point, the tracking for the cell is considered as complete. Likewise, if a feature is not assigned to a current centroid, this feature is considered a new object and is tracked as the algorithm progresses. The complete path for each feature is then stored for motility analysis.

Motion Classification: To classify the motility pattern of T. gondii in 3D space in an unsupervised fashion we implement and use the method that Fazli et. al. introduced [FSA+19]. In that work, they used an autoregressive model (AR); a linear dynamical system that encodes a Markov-based transition prediction method. The reason is that although K-means is a favorable clustering algorithm, there are a few drawbacks to it and to the conventional methods that draw them impractical. Firstly, K-means assumes Euclidian distance, but AR motion parameters are geodesics that do not reside in a Euclidean space, and secondly, K-means assumes isotropic clusters, however, although AR motion parameters may exhibit isotropy in their space, without a proper distance metric, this issue cannot be clearly examined [FSA+19].

Conclusion and Discussion

TSeg is an easy to use pipeline designed to study the motility patterns of T. gondii in 3D space. It is developed as a plugin for Napari and is equipped with a variety of deep learning based segmentation tools borrowed from PlantSeg and CellPose, making it a suitable off-the-shelf tool for applications incorporating images of cell types not limited to T. gondii. Future work on TSeg includes the expantion of implemented algorithms and tools in its preprocessing, segmentation, tracking, and clustering modules.

REFERENCES

[FRF+20] Elnaz Fazeli, Nathan H Roy, Gautier Follain, Romain F Laine, Lucas von Chamier, Pekka E Hänninen, John E Eriksson, Jean-Yves Tinevez, and Guillaume Jacquemet. Automated cell tracking using stardist and trackmate. *F1000Research*, 9, 2020. doi:10.12688/f1000research.27019.1.

[FSA+19] Mojtaba Sedigh Fazli, Rachel V Stadler, BahaaEddin Alaila, Stephen A Vella, Silvia NJ Moreno, Gary E Ward, and Shannon Quinn. Lightweight and scalable particle tracking and motion clustering of 3d cell trajectories. In *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 412–421. IEEE, 2019. doi:10.1109/dsaa.2019.00056.

[FVM+18] Mojtaba S Fazli, Stephen A Vella, Silvia NJ Moreno, Gary E Ward, and Shannon P Quinn. Toward simple & scalable 3d cell tracking. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3217–3225. IEEE, 2018. doi:10.1109/BigData.2018.8622403.

[FVMQ18] Mojtaba S Fazli, Stephen A Velia, Silvia NJ Moreno, and Shannon Quinn. Unsupervised discovery of toxoplasma gondii motility phenotypes. In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*, pages 981–984. IEEE, 2018. doi:10.1109/isbi.2018.8363735.

[KC21] Varun Kapoor and Claudia Carabaña. Cell tracking in 3d using deep learning segmentations. In *Python in Science Conference*, pages 154–161, 2021. doi:10.25080/majora-1b6fd038-014.

[KPR+21] Anuradha Kar, Manuel Petit, Yassin Refahi, Guillaume Cerutti, Christophe Godin, and Jan Traas. Assessment of deep learning algorithms for 3d instance segmentation of confocal image datasets. *bioRxiv*, 2021. URL: https://www.biorxiv.org/content/early/2021/06/10/2021.06.09.447748, arXiv:https://www.biorxiv.org/content/early/2021/06/10/2021.06.09.447748.full.pdf, doi:10.1101/2021.06.09.447748.

[LRK+14] Jacqueline Leung, Mark Rould, Christoph Konradt, Christopher Hunter, and Gary Ward. Disruption of tgphil1 alters specific parameters of toxoplasma gondii motility measured in a quantitative, three-dimensional live motility assay. *PloS one*, 9:e85763, 01 2014. doi:10.1371/journal.pone.0085763.

[SG12] Geita Saadatnia and Majid Golkar. A review on human toxoplasmosis. *Scandinavian journal of infectious diseases*, 44(11):805–814, 2012. doi:10.3109/00365548.2012.693197.

[SLE+22] Nicholas Sofroniew, Talley Lambert, Kira Evans, Juan Nunez-Iglesias, Grzegorz Bokota, Philip Winston, Gonzalo Peña-Castellanos, Kevin Yamauchi, Matthias Bussonnier, Draga Doncila Pop, Ahmet Can Solak, Ziyang Liu, Pam Wadhwa, Alister Burt, Genevieve Buckley, Andrew Sweet, Lukasz Migas, Volker Hilsenstein, Lorenzo Gaifas, Jordão Bragantini, Jaime Rodríguez-Guerra, Hector Muñoz, Jeremy Freeman, Peter Boone, Alan Lowe, Christoph Gohlke, Loic Royer, Andrea PIERRÉ, Hagai Har-Gil, and Abigail McGovern. napari: a multi-dimensional image viewer for Python, May 2022. If you use this software, please cite it using these metadata. URL: https://doi.org/10.5281/zenodo.6598542, doi:10.5281/zenodo.6598542.

[SWMP21] Carsen Stringer, Tim Wang, Michalis Michaelos, and Marius Pachitariu. Cellpose: a generalist algorithm for cellular segmentation. *Nature methods*, 18(1):100–106, 2021. doi:10.1101/2020.02.02.931238.

[TPS+17] Jean-Yves Tinevez, Nick Perry, Johannes Schindelin, Genevieve M. Hoopes, Gregory D. Reynolds, Emmanuel Laplantine, Sebastian Y. Bednarek, Spencer L. Shorte, and Kevin W. Eliceiri. Trackmate: An open and extensible platform for single-particle tracking. *Methods*, 115:80–90, 2017. Image Processing for Biologists. URL: https://www.sciencedirect.com/science/article/pii/S1046202316303346, doi:https://doi.org/10.1016/j.ymeth.2016.09.016.

[vCLJ+21] Lucas von Chamier, Romain F Laine, Johanna Jukkala, Christoph Spahn, Daniel Krentzel, Elias Nehme, Martina Lerche, Sara Hernández-Pérez, Pieta K Mattila, Eleni Karinou, et al. Democratising deep learning for microscopy with zerocostdl4mic. *Nature communications*, 12(1):1–18, 2021. doi:10.1038/s41467-021-22518-0.

[WCV+20] Adrian Wolny, Lorenzo Cerrone, Athul Vijayan, Rachele Tofanelli, Amaya Vilches Barro, Marion Louveaux, Christian Wenzl, Sören Strauss, David Wilson-Sánchez, Rena Lymbouridou, Susanne S Steigleder, Constantin Pape, Alberto Bailoni, Salva Duran-Nebreda, George W Bassel, Jan U Lohmann, Miltos Tsiantis, Fred A Hamprecht, Kay Schneitz, Alexis Maizel, and Anna Kreshuk. Accurate and versatile 3d segmentation of plant tissues at cellular resolution. *eLife*, 9:e57613, jul 2020. URL: https://doi.org/10.7554/eLife.57613, doi:10.7554/eLife.57613.

[WMV+21] Chentao Wen, Takuya Miura, Venkatakaushik Voleti, Kazushi Yamaguchi, Motosuke Tsutsumi, Kei Yamamoto, Kohei Otomo, Yukako Fujie, Takayuki Teramoto, Takeshi Ishihara, Kazuhiro Aoki, Tomomi Nemoto, Elizabeth Mc Hillman, and Koutarou D Kimura. 3DeeCellTracker, a deep learning-based pipeline for segmenting and tracking cells in 3D time lapse images. *Elife*, 10, March 2021. URL: https://doi.org/10.7554/eLife.59187, doi:10.7554/eLife.59187.

[WSH+20] Martin Weigert, Uwe Schmidt, Robert Haase, Ko Sugawara, and Gene Myers. Star-convex polyhedra for 3d object detection and segmentation in microscopy. In *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, mar 2020. URL: https://doi.org/10.1109%2Fwacv45572.2020.9093435, doi:10.1109/wacv45572.2020.9093435.

The myth of the normal curve and what to do about it

Allan Campopiano*

Index Terms—Python, R, robust statistics, bootstrapping, trimmed mean, data science, hypothesis testing

Reliance on the normal curve as a tool for measurement is almost a given. It shapes our grading systems, our measures of intelligence, and importantly, it forms the mathematical backbone of many of our inferential statistical tests and algorithms. Some even call it "God's curve" for its supposed presence in nature [Mic89].

Scientific fields that deal in explanatory and predictive statistics make particular use of the normal curve, often using it to conveniently define thresholds beyond which a result is considered statistically significant (e.g., t-test, F-test). Even familiar machine learning models have, buried in their guts, an assumption of the normal curve (e.g., LDA, gaussian naive Bayes, logistic & linear regression).

The normal curve has had a grip on us for some time; the aphorism by Cramer [Cra46] still rings true for many today:

> "Everyone believes in the [normal] law of errors, the experimenters because they think it is a mathematical theorem, the mathematicians because they think it is an experimental fact."

Many students of statistics learn that N=40 is enough to ignore the violation of the assumption of normality. This belief stems from early research showing that the sampling distribution of the mean quickly approaches normal, even when drawing from non-normal distributions—as long as samples are sufficiently large. It is common to demonstrate this result by sampling from uniform and exponential distributions. Since these look nothing like the normal curve, it was assumed that N=40 must be enough to avoid practical issues when sampling from other types of non-normal distributions [Wil13]. (Others reached similar conclusions with different methodology [Gle93].)

Two practical issues have since been identified based on this early research: (1) The distributions under study were light tailed (they did not produce outliers), and (2) statistics other than the sample mean were not tested and may behave differently. In the half century following these early findings, many important discoveries have been made—calling into question the usefulness of the normal curve [Wil13].

The following sections uncover various pitfalls one might encounter when assuming normality—especially as they relate to hypothesis testing. To help researchers overcome these problems, a new Python library for robust hypothesis testing will be introduced along with an interactive tool for robust statistics education.

* *Corresponding author: allan@deepnote.com*

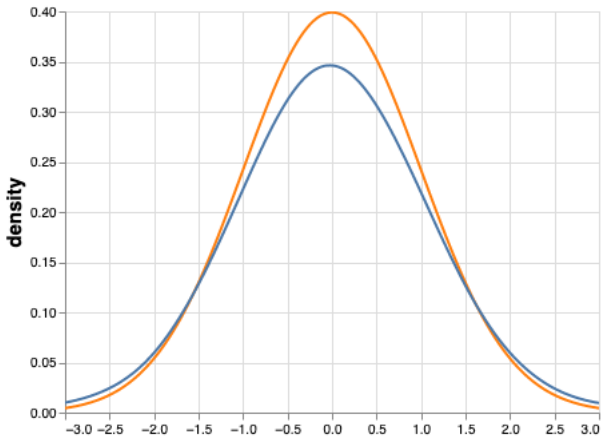


Fig. 1: *Standard normal (orange) and contaminated normal (blue). The variance of the contaminated curve is more than 10 times that of the standard normal curve. This can cause serious issues with statistical power when using traditional hypothesis testing methods.*

The contaminated normal

One of the most striking counterexamples of "N=40 is enough" is shown when sampling from the so-called contaminated normal [Tuk60][Tan82]. This distribution is also bell shaped and symmetrical but it has slightly heavier tails when compared to the standard normal curve. That is, it contains outliers and is difficult to distinguish from a normal distribution with the naked eye. Consider the distributions in Figure 1. The variance of the normal distribution is 1 but the variance of the contaminated normal is 10.9!

The consequence of this inflated variance is apparent when examining statistical power. To demonstrate, Figure 2 shows two pairs of distributions: On the left, there are two normal distributions (variance 1) and on the right there are two contaminated distributions (variance 10.9). Both pairs of distributions have a mean difference of 0.8. Wilcox [Wil13] showed that by taking random samples of N=40 from each normal curve, and comparing them with Student's t-test, statistical power was approximately 0.94. However, when following this same procedure for the contaminated groups, statistical power was only 0.25.

The point here is that even small apparent departures from normality, especially in the tails, can have a large impact on commonly used statistics. The problems continue to get worse when examining effect sizes but these findings are not discussed

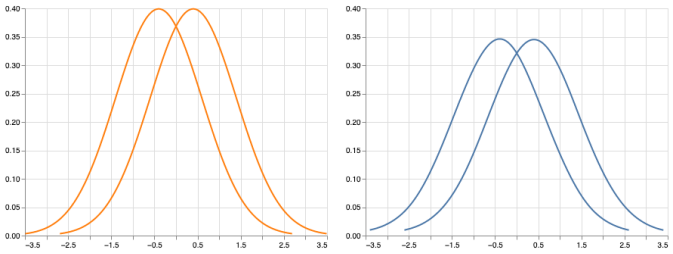


Fig. 2: *Two normal curves (left) and two contaminated normal curves (right). Despite the obvious effect sizes ($\Delta = 0.8$ for both pairs) as well as the visual similarities of the distributions, power is only ~0.25 under contamination; however, power is ~0.94 under normality (using Student's t-test).*

in this article. Interested readers should see Wilcox's 1992 paper [Wil92].

Perhaps one could argue that the contaminated normal distribution actually represents an extreme departure from normality and therefore should not be taken seriously; however, distributions that generate outliers are likely common in practice [HD82][Mic89][Wil09]. A reasonable goal would then be to choose methods that perform well under such situations and continue to perform well under normality. In addition, serious issues still exist even when examining light-tailed and skewed distributions (e.g., lognormal), and statistics other than the sample mean (e.g., T). These findings will be discussed in the following section.

Student's t-distribution

Another common statistic is the T value obtained from Student's t-test. As will be demonstrated, T is more sensitive to violations of normality than the sample mean (which has already been shown to not be robust). This is despite the fact that the t-distribution is also bell shaped, light tailed, and symmetrical—a close relative of the normal curve.

The assumption is that T follows a t-distribution (and with large samples it approaches normality). We can test this assumption by generating random samples from a lognormal distribution. Specifically, 5000 datasets of sample size 20 were randomly drawn from a lognormal distribution using SciPy's `lognorm.rvs` function. For each dataset, T was calculated and the resulting t-distribution was plotted. Figure 3 shows that the assumption that T follows a t-distribution does not hold.

With N=20, the assumption is that with a probability of 0.95, T will be between -2.09 and 2.09. However, when sampling from a lognormal distribution in the manner just described, there is actually a 0.95 probability that T will be between approximately -4.2 and 1.4 (i.e., the middle 95% of the actual t-distribution is much wider than the assumed t-distribution). Based on this result we can conclude that sampling from skewed distributions (e.g., lognormal) leads to increased Type I Error when using Student's t-test [Wil98].

> "Surely the hallowed bell-shaped curve has cracked from top to bottom. Perhaps, like the Liberty Bell, it should be enshrined somewhere as a memorial to more heroic days — Earnest Ernest, Philadelphia Inquirer. 10 November 1974. [FG81]"

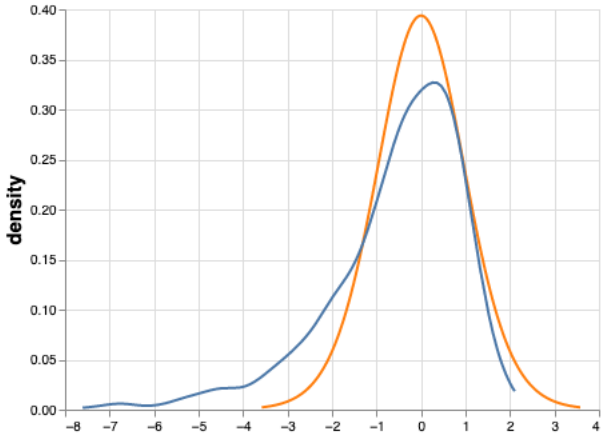


Fig. 3: *Actual t-distribution (orange) and assumed t-distribution (blue). When simulating a t-distribution based on a lognormal curve, T does not follow the assumed shape. This can cause poor probability coverage and increased Type I Error when using traditional hypothesis testing approaches.*

Modern robust methods

When it comes to hypothesis testing, one intuitive way of dealing with the issues described above would be to (1) replace the sample mean (and standard deviation) with a robust alternative and (2) use a non-parametric resampling technique to estimate the sampling distribution (rather than assuming a theoretical shape)[1]. Two such candidates are the 20% trimmed mean and the percentile bootstrap test, both of which have been shown to have practical value when dealing with issues of outliers and non-normality [CvNS18][Wil13].

The trimmed mean

The trimmed mean is nothing more than sorting values, removing a proportion from each tail, and computing the mean on the remaining values. Formally,

- Let $X_1 ... X_n$ be a random sample and $X_{(1)} \leq X_{(2)} ... \leq X_{(n)}$ be the observations in ascending order
- The proportion to trim is $\gamma (0 \leq \gamma \leq .5)$
- Let $g = \lfloor \gamma n \rfloor$. That is, the proportion to trim multiplied by n, rounded down to the nearest integer

Then, in symbols, the trimmed mean can be expressed as follows:

$$\bar{X}_t = \frac{X_{(g+1)} + ... + X_{(n-g)}}{n - 2g}$$

If the proportion to trim is 0.2, more than twenty percent of the values would have to be altered to make the trimmed mean arbitrarily large or small. The sample mean, on the other hand, can be made to go to $\pm\infty$ (arbitrarily large or small) by changing a single value. The trimmed mean is more robust than the sample mean in all measures of robustness that have been studied [Wil13]. In particular the 20% trimmed mean has been shown to have practical value as it avoids issues associated with the median (not discussed here) and still protects against outliers.

1. Another option is to use a parametric test that assumes a different underlying model.

The percentile bootstrap test

In most traditional parametric tests, there is an assumption that the sampling distribution has a particular shape (normal, f-distribution, t-distribution, etc). We can use these distributions to test the null hypothesis; however, as discussed, the theoretical distributions are not always approximated well when violations of assumptions occur. Non-parametric resampling techniques such as bootstrapping and permutation tests build empirical sampling distributions, and from these, one can robustly derive p-values and CIs. One example is the percentile bootstrap test [Efr92][TE93].

The percentile bootstrap test can be thought of as an algorithm that uses the data at hand to estimate the underlying sampling distribution of a statistic (pulling yourself up by your own bootstraps, as the saying goes). This approach is in contrast to traditional methods that assume the sampling distribution takes a particular shape). The percentile boostrap test works well with small sample sizes, under normality, under non-normality, and it easily extends to multi-group tests (ANOVA) and measures of association (correlation, regression). For a two-sample case, the steps to compute the percentile bootstrap test can be described as follows:

1) Randomly resample with replacement n values from group one
2) Randomly resample with replacement n values from group two
3) Compute $\bar{X}_1 - \bar{X}_2$ based on you new sample (the mean difference)
4) Store the difference & repeat steps 1-3 many times (say, 1000)
5) Consider the middle 95% of all differences (the confidence interval)
6) If the confidence interval contains zero, there is no statistical difference, otherwise, you can reject the null hypothesis (there is a statistical difference)

Implementing and teaching modern robust methods

Despite over a half a century of convincing findings, and thousands of papers, robust statistical methods are still not widely adopted in applied research [EHM08][Wil98]. This may be due to various *false* beliefs. For example,

- Classical methods are robust to violations of assumptions
- Correcting non-normal distributions by transforming the data will solve all issues
- Traditional non-parametric tests are suitable replacements for parametric tests that violate assumptions

Perhaps the most obvious reason for the lack of adoption of modern methods is a lack of easy-to-use software and training resources. In the following sections, two resources will be presented: one for implementing robust methods and one for teaching them.

Robust statistics for Python

Hypothesize is a robust null hypothesis significance testing (NHST) library for Python [CW20]. It is based on Wilcox's WRS package for R which contains hundreds of functions for computing robust measures of central tendency and hypothesis testing. At the time of this writing, the WRS library in R contains many more functions than Hypothesize and its value to researchers who use inferential statistics cannot be understated. WRS is best experienced in tandem with Wilcox's book "Introduction to Robust Estimation and Hypothesis Testing".

Hypothesize brings many of these functions into the open-source Python library ecosystem with the goal of lowering the barrier to modern robust methods—even for those who have not had extensive training in statistics or coding. With modern browser-based notebook environments (e.g., Deepnote), learning to use Hypothesize can be relatively straightforward. In fact, every statistical test listed in the docs is associated with a hosted notebook, pre-filled with sample data and code. But certainly, simply `pip install Hypothesize` to use Hypothesize in any environment that supports Python. See van Noordt and Willoughby [vNW21] and van Noordt et al. [vNDTE22] for examples of Hypothesize being used in applied research.

The API for Hypothesize is organized by single- and two-factor tests, as well as measures of association. Input data for the groups, conditions, and measures are given in the form of a Pandas DataFrame [pdt20][WM10]. By way of example, one can compare two independent groups (e.g., placebo versus treatment) using the 20% trimmed mean and the percentile bootstrap test, as follows (note that Hypothesize uses the naming conventions found in WRS):

```
from hypothesize.utilities import trim_mean
from hypothesize.compare_groups_with_single_factor \
    import pb2gen

results = pb2gen(df.placebo, df.treatment, trim_mean)
```

As shown below, the results are returned as a Python dictionary containing the p-value, confidence intervals, and other important details.

```
{
'ci': [-0.22625614592148624, 0.06961754796950131],
'est_1': 0.43968438076483285,
'est_2': 0.5290985245430996,
'est_dif': -0.08941414377826673,
'n1': 50,
'n2': 50,
'p_value': 0.27,
'variance': 0.005787027326924963
}
```

For measuring associations, several options exist in Hypothesize. One example is the Winsorized correlation which is a robust alternative to Pearson's R. For example,

```
from hypothesize.measuring_associations import wincor

results = wincor(df.height, df.weight, tr=.2)
```

returns the Winsorized correlation coefficient and other relevant statistics:

```
{
'cor': 0.08515087411576182,
'nval': 50,
'sig': 0.558539575073185,
'wcov': 0.004207827245660796
}
```

A case study using real-world data

It is helpful to demonstrate that robust methods in Hypothesize (and in other libraries) can make a practical difference when dealing with real-world data. In a study by Miller on sexual attitudes, 1327 men and 2282 women were asked how many sexual

partners they desired over the next 30 years (the data are available from Rand R. Wilcox's site). When comparing these groups using Student's t-test, we get the following results:

```
{
'ci': [-1491.09,  4823.24],
't_value': 1.035308,
'p_value': 0.300727
}
```

That is, we fail to reject the null hypothesis at the $\alpha = 0.05$ level using Student's test for independent groups. However, if we switch to a robust analogue of the t-test, one that utilizes bootstrapping and trimmed means, we can indeed reject the null hypothesis. Here are the corresponding results from Hypothesize's `yuenbt` test (based on [Yue74]):

```
from hypothesize.compare_groups_with_single_factor \
    import yuenbt

results = yuenbt(df.males, df.females,
    tr=.2, alpha=.05)
```

```
{
'ci': [1.41, 2.11],
'test_stat': 9.85,
'p_value': 0.0
}
```

The point here is that robust statistics can make a practical difference with real-world data (even when N is considered large). Many other examples of robust statistics making a practical difference with real-world data have been documented [HD82][Wil09][Wil01].

It is important to note that robust methods may also fail to reject when a traditional test rejects (remember that traditional tests can suffer from increased Type I Error). It is also possible that both approaches yield the same or similar conclusions. The exact pattern of results depends largely on the characteristics of the underlying population distribution. To be able to reason about how robust statistics behave when compared to traditional methods the robust statistics simulator has been created and is described in the next section.

Robust statistics simulator

Having a library of robust statistical functions is not enough to make modern methods commonplace in applied research. Educators and practitioners still need intuitive training tools that demonstrate the core issues surrounding classical methods and how robust analogues compare.

As mentioned, computational notebooks that run in the cloud offer a unique solution to learning beyond that of static textbooks and documentation. Learning can be interactive and exploratory since narration, visualization, widgets (e.g., buttons, slider bars), and code can all be experienced in a ready-to-go compute environment—with no overhead related to local environment setup.

As a compendium to Hypothesize, and a resource for understanding and teaching robust statistics in general, the robust statistics simulator repository has been developed. It is a notebook-based collection of interactive demonstrations aimed at clearly and visually explaining the conditions under which classic methods fail relative to robust methods. A hosted notebook with the rendered visualizations of the simulations can be accessed here. and seen in Figure 4. Since the simulations run in the browser and require very little understanding of code, students and teachers can easily onboard to the study of robust statistics.

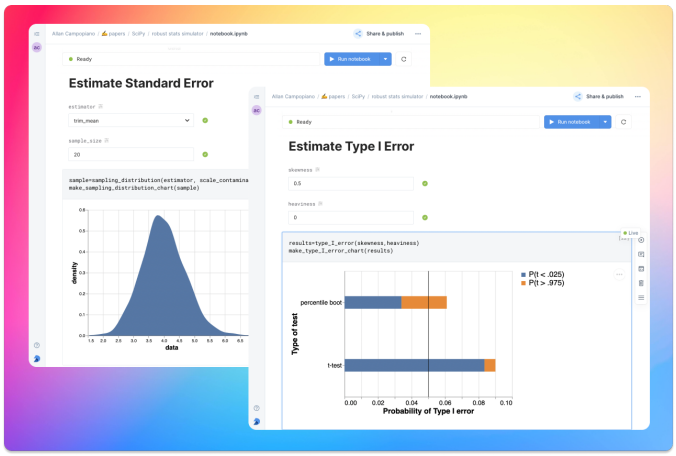


Fig. 4: *An example of the robust stats simulator in Deepnote's hosted notebook environment. A minimalist UI can lower the barrier-to-entry to robust statistics concepts.*

The robust statistics simulator allows users to interact with the following parameters:

- Distribution shape
- Level of contamination
- Sample size
- Skew and heaviness of tails

Each of these characteristics can be adjusted independently in order to compare classic approaches to their robust alternatives. The two measures that are used to evaluate the performance of classic and robust methods are the standard error and Type I Error.

Standard error is a measure of how much an estimator varies across random samples from our population. We want to choose estimators that have a low standard error. Type I Error is also known as False Positive Rate. We want to choose methods that keep Type I Error close to the nominal rate (usually 0.05). The robust statistics simulator can guide these decisions by providing empirical evidence as to why particular estimators and statistical tests have been chosen.

Conclusion

This paper gives an overview of the issues associated with the normal curve. The concern with traditional methods, in terms of robustness to violations of normality, have been known for over a half century and modern alternatives have been recommended; however, for various reasons that have been discussed, modern robust methods have not yet become commonplace in applied research settings.

One reason is the lack of easy-to-use software and teaching resources for robust statistics. To help fill this gap, Hypothesize, a peer-reviewed and open-source Python library was developed. In addition, to help clearly demonstrate and visualize the advantages of robust methods, the robust statistics simulator was created. Using these tools, practitioners can begin to integrate robust statistical methods into their inferential testing repertoire.

Acknowledgements

The author would like to thank Karlynn Chan and Rand R. Wilcox as well as Elizabeth Dlha and the entire Deepnote team for their support of this project. In addition, the author would like to thank Kelvin Lee for his insightful review of this manuscript.

REFERENCES

[Cra46] Harold Cramer. Mathematical methods of statistics, princeton univ. *Press, Princeton, NJ*, 1946. URL: https://books.google.ca/books?id=CRTKKaJO0DYC.

[CvNS18] Allan Campopiano, Stefon JR van Noordt, and Sidney J Segalowitz. Statslab: An open-source eeg toolbox for computing single-subject effects using robust statistics. *Behavioural Brain Research*, 347:425–435, 2018. doi:10.1016/j.bbr.2018.03.025.

[CW20] Allan Campopiano and Rand R. Wilcox. Hypothesize: Robust statistics for python. *Journal of Open Source Software*, 5(50):2241, 2020. doi:10.21105/joss.02241.

[Efr92] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992. doi:10.1007/978-1-4612-4380-9_41.

[EHM08] David M Erceg-Hurn and Vikki M Mirosevich. Modern robust statistical methods: an easy way to maximize the accuracy and power of your research. *American Psychologist*, 63(7):591, 2008. doi:10.1037/0003-066X.63.7.591.

[FG81] Joseph Fashing and Ted Goertzel. The myth of the normal curve a theoretical critique and examination of its role in teaching and research. *Humanity & Society*, 5(1):14–31, 1981. doi:10.1177/016059768100500103.

[Gle93] John R Gleason. Understanding elongation: The scale contaminated normal family. *Journal of the American Statistical Association*, 88(421):327–337, 1993. doi:10.1080/01621459.1993.10594325.

[HD82] MaryAnn Hill and WJ Dixon. Robustness in real life: A study of clinical laboratory data. *Biometrics*, pages 377–396, 1982. doi:10.2307/2530452.

[Mic89] Theodore Micceri. The unicorn, the normal curve, and other improbable creatures. *Psychological bulletin*, 105(1):156, 1989. doi:10.1037/0033-2909.105.1.156.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[Tan82] WY Tan. Sampling distributions and robustness of t, f and variance-ratio in two samples and anova models with respect to departure from normality. *Comm. Statist.-Theor. Meth.*, 11:2485–2511, 1982. URL: https://pascal-francis.inist.fr/vibad/index.php?action=getRecordDetail&idt=PASCAL83X0380619.

[TE93] Robert J Tibshirani and Bradley Efron. An introduction to the bootstrap. *Monographs on statistics and applied probability*, 57:1–436, 1993. URL: https://books.google.ca/books?id=gLlpIUxRntoC.

[Tuk60] J. W. Tukey. A survey of sampling from contaminated distributions. *Contributions to Probability and Statistics*, pages 448–485, 1960. URL: https://ci.nii.ac.jp/naid/20000755025/en/.

[vNDTE22] Stefon van Noordt, James A Desjardins, BASIS Team, and Mayada Elsabbagh. Inter-trial theta phase consistency during face processing in infants is associated with later emerging autism. *Autism Research*, 15(5):834–846, 2022. doi:10.1002/aur.2701.

[vNW21] Stefon van Noordt and Teena Willoughby. Cortical maturation from childhood to adolescence is reflected in resting state eeg signal complexity. *Developmental cognitive neuroscience*, 48:100945, 2021. doi:10.1016/j.dcn.2021.100945.

[Wil92] Rand R Wilcox. Why can methods for comparing means have relatively low power, and what can you do to correct the problem? *Current Directions in Psychological Science*, 1(3):101–105, 1992. doi:10.1111/1467-8721.ep10768801.

[Wil98] Rand R Wilcox. How many discoveries have been lost by ignoring modern statistical methods? *American Psychologist*, 53(3):300, 1998. doi:10.1037/0003-066X.53.3.300.

[Wil01] Rand R Wilcox. *Fundamentals of modern statistical methods: Substantially improving power and accuracy*, volume 249. Springer, 2001. URL: https://link.springer.com/book/10.1007/978-1-4757-3522-2.

[Wil09] Rand R Wilcox. Robust ancova using a smoother with bootstrap bagging. *British Journal of Mathematical and Statistical Psychology*, 62(2):427–437, 2009. doi:10.1348/000711008X325300.

[Wil13] Rand R Wilcox. *Introduction to robust estimation and hypothesis testing*. Academic press, 2013. doi:10.1016/c2010-0-67044-1.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a.

[Yue74] Karen K Yuen. The two-sample trimmed t for unequal population variances. *Biometrika*, 61(1):165–170, 1974. doi:10.2307/2334299.

Python for Global Applications: teaching scientific Python in context to law and diplomacy students

Anna Haensch[‡§*], Karin Knudson[‡§]

Abstract—For students across domains and disciplines, the message has been communicated loud and clear: data skills are an essential qualification for today's job market. This includes not only the traditional introductory stats coursework but also machine learning, artificial intelligence, and programming in Python or R. Consequently, there has been significant student-initiated demand for data analytic and computational skills sometimes with very clear objectives in mind, and other times guided by a vague sense of "the work I want to do will require this." Now we have options. If we train students using "black box" algorithms without attending to the technical choices involved, then we run the risk of unleashing practitioners who might do more harm than good. On the other hand, courses that completely unpack the "black box" can be so steeped in theory that the barrier to entry becomes too high for students from social science and policy backgrounds, thereby excluding critical voices. In sum, both of these options lead to a pitfall that has gained significant media attention over recent years: the harms caused by algorithms that are implemented without sufficient attention to human context. In this paper, we - two mathematicians turned data scientists - present a framework for teaching introductory data science skills in a highly contextualized and domain flexible environment. We will present example course outlines at the semester, weekly, and daily level, and share materials that we think hold promise.

Index Terms—computational social science, public policy, data science, teaching with Python

Introduction

As data science continues to gain prominence in the public eye, and as we become more aware of the many facets of our lives that intersect with data-driven technologies and policies every day, universities are broadening their academic offerings to keep up with what students and their future employers demand. Not only are students hoping to obtain more hard skills in data science (e.g. Python programming experience), but they are interested in applying tools of data science across domains that haven't historically been part of the quantitative curriculum. The Master of Arts in Law and Diplomacy (MALD) is the flagship program of the Fletcher School of Law and International Diplomacy at Tufts University. Historically, the program has contained core elements of quantitative reasoning with a focus on business, finance, and international development, as is typical in graduate programs in international relations. Like academic institutions more broadly, the students and faculty at the Fletcher School are eager to seize upon our current data moment to expand their quantitative offerings. With this in mind, The Fletcher School reached out to the co-authors to develop a course in data science, situated in the context of international diplomacy.

In response, we developed the (Python-based) course, *Data Science for Global Applications*, which had its inaugural offering in the Spring semester of 2022. The course had 30 enrolled Fletcher School students, primarily from the MALD program. When the course was announced we had a flood of interest from Fletcher students who were extremely interested in broadening their studies with this course. With a goal of keeping a close interactive atmosphere we capped enrollment at 30. To inform the direction of our course, we surveyed students on their background in programming (see Fig. 1) and on their motivations for learning data science (see Fig 2). Students reported only very limited experience with programming - if any at all - with that experience primarily in Excel and Tableau. Student motivations varied, but the goal to get a job where they were able to make a meaningful social impact was the primary motivation.

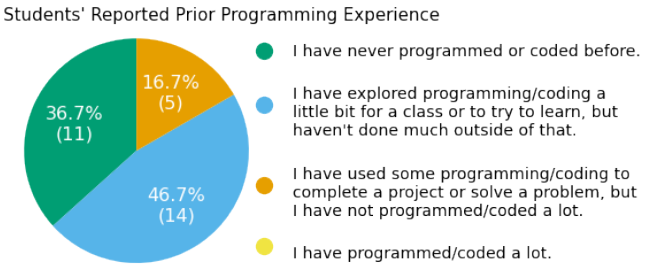


Fig. 1: *The majority of the 30 students enrolled in the course had little to no programming experience, and none reported having "a lot" of experience. Those who did have some experience were most likely to have worked in Excel or Tableau.*

The MALD program, which is interdisciplinary by design, provides ample footholds for domain specific data science. Keeping this in mind, as a throughline for the course, each student worked to develop their own quantitative policy project. Coursework and discussions were designed to move this project forward from initial policy question, to data sourcing and visualizing, and eventually to modeling and analysis.

In what follows we will describe how we structured our course with the goal of empowering beginner programmers to use Python for data science in the context of international relations

* *Corresponding author: anna.haensch@tufts.edu*
‡ *Tufts University*
§ *Data Intensive Studies Center*

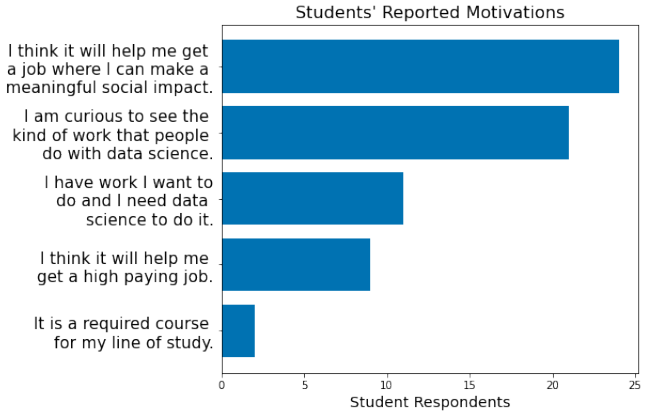


Fig. 2: *The 30 enrolled students were asked to indicate which were relevant motivations for taking the course. Curiosity and a desire to make a meaningful social impact were among the top motivations our students expressed.*

and diplomacy. We will also share details about course content and structure, methods of assessment, and Python programming resources that we deployed through Google Colab. All of the materials described here can be found on the public course page https://karink520.github.io/data-science-for-global-applications/.

Course Philosophy and Goals

Our high level goals for the course were i) to empower students with the skills to gain insight from data using Python and ii) to deepen students' understanding of how the use of data science affects society. As we sought to achieve these high level goals within the limited time scope of a single semester, the following core principles were essential in shaping our course design. Below, we briefly describe each of these principles and share some examples of how they were reflected in the course structure. In a subsequent section we will more precisely describe the content of the course, whereupon we will further elaborate on these principles and share instructional materials. But first, our core principles:

Connecting the Technical and Social

To understand the impact of data science on the world (and the potential policy implications of such impact), it helps to have hands-on practice with data science. Conversely, to effectively and ethically practice data science, it is important to understand how data science lives in the world. Thus, the "hard" skills of coding, wrangling data, visualizing, and modeling are best taught intertwined with a robust study of ways in which data science is used and misused.

There is an increasing need to educate future policy-makers with knowledge of how data science algorithms can be used and misused. One way to approach meeting this need, especially for students within a less technically-focused program, would be to teach students about how algorithms can be used without actually teaching them to use algorithms. However, we argue that students will gain a deeper understanding of the societal and ethical implications of data science if they also have practical data science skills. For example, a student could gain a broad understanding of how biased training data might lead to biased algorithmic predictions, but such understanding is likely to be deeper and more memorable when a student has actually practiced training a model using different training data. Similarly, someone might understand in the abstract that the way the handling of missing data can substantially affect the outcome of an analysis, but will likely have a stronger understanding if they have had to consider how to deal with missing data in their own project.

We used several course structures to support connecting data science and Python "skills" with their context. Students had readings and journaling assignments throughout the semester on topics that connected data science with society. In their journal responses, students were asked to connect the ideas in the reading to their other academic/professional interests, or ideas from other classes with the following prompt:

> *Your reflection should be a 250-300 word narrative. Be sure to tie the reading back into your own studies, experiences, and areas of interest. For each reading, come up with 1-2 discussion questions based on the concepts discussed in the readings. This can be a curiosity question, where you're interested in finding out more, a critical question, where you challenge the author's assumptions or decisions, or an application question, where you think about how concepts from the reading would apply to a particular context you are interested in exploring.*[1]

These readings (highlighted in gray in Fig 3), assignments, and the related in-class discussions were interleaved among Python exercises meant to give students practice with skills including manipulating DataFrames in pandas [The22], [Mck10], plotting in Matplotlib [Hun07] and seaborn [Was21], mapping with GeoPandas [Jor21], and modeling with scikit-learn [Ped11]. Student projects included a thorough data audit component requiring students to explore data sources and their human context in detail. Precise details and language around the data audit can be found on the course website.

Managing Fears & Concerns Through Supported Programming

We surmised that students who are new to programming and possibly intimidated by learning the unfamiliar skill would do well in an environment that included plenty of what we call *supported programming* - that is, practicing programming in class with immediate access to instructor and peer support.

In the pre-course survey we created, many students identified concerns about their quantitative preparation, whether they would be able to keep up with the course, and how hard programming might be. We sought to acknowledge these concerns head-on, assure students of our full confidence in their ability to master the material, and provide them with all the resources they needed to succeed.

A key resource to which we thought all students needed access was instructor attention. In addition to keeping the class size capped at 30 people, with both co-instructors attending all course meetings, we structured class time to maximize the time students spent actually doing data science in class. We sought to keep demonstrations short, and intersperse them with coding exercises so that students could practice with new ideas right away. Our Colab notebooks included in the course materials show one way that we wove student practice time throughout. Drawing insight from social practice theory of learning (e.g. [Eng01], [Pen16]), we sought to keep in mind how individual practice and learning pathways develop in relation to their particular social and

1. This journaling prompt was developed by our colleague Desen Ozkan at Tufts University.

institutional context. Crucially, we devoted a great deal of in-class time to students doing data science, and a great deal of energy into making this practice time a positive and empowering social experience. During student practice time, we were circulating throughout the room, answering student questions and helping students to problem solve and debug, and encouraging students to work together and help each other. A small organizational change we made in the first weeks of the semester that proved to have outsized impact was moving our office hours to hold them directly after class in an almost-adjacent room, to make it as easy as possible for students to attend office hours. Students were vocal in their appreciation of office hours.

We contend that the value of supported programming time is two-fold. First, it helps beginning programmers learn more quickly. While learning to code necessarily involves challenges, students new to a language can sometimes struggle for an unproductively long time on things like simple syntax issues. When students have help available, they can move forward from minor issues faster and move more efficiently into building a meaningful understanding. Secondly, supported programming time helps students to understand that they are not alone in the challenges they are facing in learning to program. They can see other students learning and facing similar challenges, can have the empowering experience of helping each other out, and when asking for help can notice that even their instructors sometimes rely on resources like StackOverflow. An unforeseen benefit we believe co-teaching had was to give us as instructors the opportunity to consult with each other during class time and share different approaches. These instructor interactions modeled for students how even as experienced practitioners of data science, we too were constantly learning.

Lastly, a small but (we thought) important aspect of our setup was teaching students to set up a computing environment on their own laptops, with Python, conda [Ana16], and JupyterLab [Pro22]. Using the command line and moving from an environment like Google Colab to one's own computer can both present significant barriers, but doing so successfully can be an important part of helping students feel like 'real' programmers. We devoted an entire class period to helping students with installation and setup on their own computers.

We considered it an important measure of success how many students told us at the end of the course that the class had helped them overcome sometimes longstanding feelings that technical skills like coding and modeling were not for them.

Leveraging Existing Strengths To Enhance Student Ownership

Even as beginning programmers, students are capable of creating a meaningful policy-related data science project within the semester, starting from formulating a question and finding relevant datasets. Working on the project throughout the semester (not just at the end) gave essential context to data science skills as students could translate into what an idea might mean for "their" data. Giving students wide leeway in their project topic allowed the project to be a point of connection between new data science skills and their existing domain knowledge. Students chose projects within their particular areas of interest or expertise, and a number chose to additionally connect their project for this course to their degree capstone project.

Project benchmarks were placed throughout the semester (highlighted in green in Fig 3) allowing students a concrete way to develop their new skills in identifying datasets, loading and preparing data for exploratory data analysis, visualizing and annotating data, and finally modeling and analyzing data. All of this was done with the goal of answering a policy question developed by the student, allowing the student to flex some domain expertise to supplement the (sometimes overwhelming!) programmatic components.

Our project explicitly required that students find two datasets of interest and merge them for the final analysis. This presented both logistical and technical challenges. As one student pointed out after finally finding open data: hearing people talk about the need for open data is one thing, but you really realize what that means when you've spent weeks trying to get access to data that you know exists. Understanding the provenance of the data they were working with helped students assess the biases and limitations, and also gave students a strong sense of ownership over their final projects. An unplanned consequence of the broad scope of the policy project was that we, the instructors, learned nearly as much about international diplomacy as the students learned about programming and data science, a bidirectional exchange of knowledge that we surmised to have contributed to student feeling of empowerment and a positive class environment.

Course Structure

We broke the course into three modules, each with focused reading/journaling topics, Python exercises, and policy project benchmarks: (i) getting and cleaning data, (ii) visualizing data, and (iii) modeling data. In what follows we will describe the key goals of each module and highlight the readings and exercises that we compiled to work towards these goals.

Getting and Cleaning Data

Getting, cleaning, and wrangling data typically make up a significant proportion of the time involved in a data science project. Therefore, we devoted significant time in our course to learning these skills, focusing on loading and manipulating data using pandas. Key skills included loading data into a pandas DataFrame, working with missing data, and slicing, grouping, and merging DataFrames in various ways. After initial exposure and practice with example datasets, students applied their skills to wrangling the diverse and sometimes messy and large datasets that they found for their individual projects. Since one requirement of the project was to integrate more than one dataset, merging was of particular importance.

During this portion of the course, students read and discussed Boyd and Crawford's *Critical Questions for Big Data* [Boy12] which situates big data in the context of knowledge itself and raises important questions about access to data and privacy. Additional readings included selected chapters from D'Ignazio and Klein's *Data Feminism* [Dig20] which highlights the importance of what we choose to count and what it means when data is missing.

Visualizing Data

A fundamental component to communicating findings from data is well-executed data visualization. We chose to place this module in the middle of the course, since it was important that students have a common language for interpreting and communicating their analysis before moving to the more complicated aspects of data modeling. In developing this common language, we used Wilke's *Fundamentals of Data Visualization* [Wil19] and Cairo's *How*

Week 1	**Intro to Python and Colab Notebooks**	**Lists, For Loops and If/Else Statements**	Getting and Cleaning Data
Reading/Discussion:	*Critical Questions for Big Data,* Crawford, K. & Boyd, D.	Demo and coding practice with Colab	
Assessment:			
Week 2	**Intro to Dataframes and Pandas**	**Data Procurement and Management**	
Reading/Discussion:	*Data Feminism: The Power Chapter,* D'Ignazio, K & Klein, L.	Guest Presentation by Research Data Librarian	
Assessment:	Project check-in: Policy Question		
Week 3	**Basics of Data Preparation and Cleaning**	**Methods for Dealing with Missing Data**	
Reading/Discussion:	Loading and manipulating dataframes with Pandas	When to impute missing data and how.	
Assessment:	Project check-in: Datasets of Interest		
Week 4	**Setting Up Your Local Python Environment**	**Installation followup and Jupyter Workflow**	
Reading/Discussion:	*Data Feminism: Collect, Analyze, Imagine, Teach,* D'Ignazio, K & Klein, L.	Loading and saving data for policy project	
Assessment:	Project check-in: Draft Data Audit	Python Exercise: Exploratory Data Analysis	
Week 5	**Basic Charts with Matplotlib**	**Principles of Data Visualization**	Visualizing Data
Reading/Discussion:	*Fundamentals of Data Visualization,* Wilke, C.	Sharing visualizations found "in the wild."	
Assessment:			
Week 6	**Mapping with Python and Geopandas**	**Mapping with ArcGIS**	
Reading/Discussion:	Plotting demos and planning draft visualizations for policy project.	Guest Presentation by GIS Specialist	
Assessment:			
Week 7	**Plot Customization and Annotation**	**Sharing draft data visualizations**	
Reading/Discussion:	*How Charts Lie,* Cairo, A. (Ch. 1-2)	Draft visualization sharing and peer feedback	
Assessment:		Python Exercise: Data Visualization	
Week 8	**Basic Descriptive Statistics with Pandas**	**Clustering with SciKit-Learn: Gaussian Mixture Models**	Modeling Data
Reading/Discussion:	Computing means, medians, modes and variance.	Model fundamentals, demo, and practice time.	
Assessment:	Project check-in: Draft data visualization		
Week 9	**Classification with SciKit-Learn: Logistic Regresion**	**Traininng and Assessing Model Performance**	
Reading/Discussion:	*Weapons of Math Destruction,* O'Neil, C. (Ch. 1)	Train/Test Split, Model Accuracy, Confusion Matrices	
Assessment:			
Week 10	**Classification with SciKit-Learn: Decision Trees, Random Forests**	**Supervised vs. Unsupervised Learning**	
Reading/Discussion:	Model fundamentals, demo, and practice time.	Choosing and planning model class for policy project	
Assessment:			
Week 11	**Regression with SciKit-Learn: Linear Regression**	**Interpreting Model Output**	
Reading/Discussion:	*Gender Shades,* Buolamwini, J. & Gebru, T.	Understanding coefficients and feature importance	
Assessment:			
Week 12	**Supported Working Time for Final Project**	**Supported Working Time for Final Project**	
Reading/Discussion:	Planning and executing data modeling and visualization.	Planning and executing data modeling and visualization.	
Assessment:		Python Exercise: Clustering, Regression and Classification	
Week 13	Project Presentations	Project Presentations	

Fig. 3: *Course outline for a 13-week semester with two 70 minute instructional blocks each week. Course readings are highlighted in gray and policy project benchmarks are highlighted in green.*

Chart's Lie [Cai19] as a backbone for this section of the course. In addition to reading the text materials, students were tasked with finding visualizations "in the wild," both good and bad. Course discussions centered on the found visualizations, with Wilke and Cairo's writings as a common foundation. From the readings and discussions, students became comfortable with the language and taxonomy around visualizations and began to develop a better appreciation of what makes a visualization compelling and readable. Students were able to formulate a plan about how they could best visualize their data. The next task was to translate these plans into Python.

To help students gain a level of comfort with data visualization in Python, we provided instruction and examples of working with a variety of charts using Matplotlib and seaborn, as well as maps and choropleths using GeoPandas, and assigned students programming assignments that involved writing code to create a visualization matching one in an image. With that practical grounding, students were ready to visualize their own project data using Python. Having the concrete target of how a student wanted their visualization to look seemed to be a motivating starting point from which to practice coding and debugging. We spent several class periods on supported programming time for students to develop their visualizations.

Working on building the narratives of their project and developing their own visualizations in the context of the course readings gave students a heightened sense of attention to detail. During one day of class when students shared visualizations and gave feedback to one another, students commented and inquired about incredibly small details of each others' presentations, for example, how to adjust y-tick alignment on a horizontal bar chart. This sort of tiny detail is hard to convey in a lecture, but gains outsized importance when a student has personally wrestled with it.

Modeling Data

In this section we sought to expose students to introductory approaches in each of regression, classification, and clustering

in Python. Specifically, we practiced using scikit-learn to work with linear regression, logistic regression, decision trees, random forests, and gaussian mixture models. Our focus was not on the theoretical underpinnings of any particular model, but rather on the kinds of problems that regression, classification, or clustering models respectively, are able to solve, as well as some basic ideas about model assessment. The uniform and approachable scikit-learn API [Bui13] was crucial in supporting this focus, since it allowed us to focus less on syntax around any one model, and more on the larger contours of modeling, with all its associated promise and perils. We spent a good deal of time building an understanding of train-test splits and their role in model assessment.

Student projects were required to include a modeling component. Just the process of deciding which of regression, classification, or clustering were appropriate for a given dataset and policy question is highly non-trivial for beginners. The diversity of student projects and datasets meant students had to grapple with this decision process in its full complexity. We were delighted by the variety of modeling approaches students used in their projects, as well as by students' thoughtful discussions of the limitations of their analysis.

To accompany this section of the course, students were assigned readings focusing on some of the societal impacts of data modeling and algorithms more broadly. These readings included a chapter from O'Neil's *Weapons of Math Destruction* [One16] as well as Buolamwini and Gebru's *Gender Shades* [Buo18]. Both of these readings emphasize the capacity of algorithms to exacerbate inequalities and highlight the importance of transparency and ethical data practices. These readings resonated especially strongly with our students, many of whom had recently taken courses in cyber policy and ethics in artificial intelligence.

Assessments

Formal assessment was based on four components, already alluded to throughout this note. The largest was the ongoing policy project which had benchmarks with rolling due dates throughout the semester. Moreover, time spent practicing coding skills in class was often done in service of the project. For example, in week 4, when students learned to set up their local computing environments, they also had time to practice loading, reading, and saving data files associated with their chosen project datasets. This brought challenges, since often students sitting side-by-side were dealing with different operating systems and data formats. But from this challenge emerged many organic conversations about file types and the importance of naming conventions. The rubric for the final project is shown in Fig 4.

The policy project culminated with in-class "micro presentations" and a policy paper. We dedicated two days of class in week 13 for in-class presentations, for which each student presented one slide consisting of a descriptive title, one visualization, and several "key takeaways" from the project. This extremely restrictive format helped students to think critically about the narrative information conveyed in a visualization, and was designed to create time for robust conversation around each presentation.

In addition to the policy project, each of the three course modules also had an associated set of Python exercises (available on the course website). Students were given ample time both in and out of class to ask questions about the exercises. Overall, these exercises proved to be the most technically challenging component of the course, but we invited students to resubmit after an initial round of grading.

And finally, to supplement the technical components of the course we also had readings with associated journal entries submitted at a cadence of roughly two per module. Journal prompts are described above and available on the course website.

Conclusion

Various listings of key competencies in data science have been proposed [NAS18]. For example, [Dev17] suggests the following pillars for an undergraduate data science curriculum: computational and statistical thinking, mathematical foundations, model building and assessment, algorithms and software foundation, data curation, and knowledge transference—communication and responsibility. As we sought to contribute to the training of data-science informed practitioners of international relations, we focused on helping students build an initial competency especially in the last four of these.

We can point to several key aspects of the course that made it successful. Primary among them was the fact that the majority of class time was spent in supported programming. This means that students were able to ask their instructors or peers as soon as questions arose. Novice programmers who aren't part of a formal computer science program often don't have immediate access to the resources necessary to get "unstuck." for the novice programmer, even learning *how* to google technical terms can be a challenge. This sort of immediate debugging and feedback helped students remain confident and optimistic about their projects. This was made all the more effective since we were co-teaching the course and had double the resources to troubleshoot. Co-teaching also had the unforeseen benefit of making our classroom a place where the growth mindset was actively modeled and nurtured: where one instructor wasn't able to answer a question, the other instructor often could. Finally, it was precisely the motivation of learning data science in context that allowed students to maintain a sense of ownership over their work and build connections between their other courses.

Learning programming from the ground up is difficult. Students arrive excited to learn, but also nervous and occasionally heavy with the baggage they carry from prior experience in quantitative courses. However, with a sufficient supported learning environment it's possible to impart relevant skills. It was a measure of the success of the course how many students told us that the course had helped them overcome negative prior beliefs about their ability to code. Teaching data science skills in context and with relevant projects that leverage students' existing expertise and outside reading situates the new knowledge in a place that feels familiar and accessible to students. This contextualization allows students to gain some mastery while simultaneously playing to their strengths and interests.

References

[Ana16] Anaconda Software Distribution. *Computer software. Vers. 2-2.4.0.* Anaconda, Nov. 2016. Web. https://anaconda.com.

[Boy12] Boyd, Danah, and Kate Crawford. *Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon*. Information, communication & society 15.5 (2012):662-679. https://doi.org/10.1080/1369118X.2012.678878

[Bui13] Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae et al. *API design for machine learning software: experiences from the scikit-learn project*. arXiv preprint arXiv:1309.0238 (2013).

Core Competency	What Success Looks Like
Data relevance and integration	Effectively integrates two or more data sources that are highly relevant to the policy question at hand.
Exploration of the data and its provenance	The source of the data is thoroughly discussed, and unresolved questions about the data collection and/or source are clearly identified.
Visual communication	The report uses a variety of different kinds of visualizations that follow principles of effective design and that display and highlight important and relevant features of the data that address the policy question. Moreover, the choice of visualizations are suitable for the aspects of the data being highlighted.
Data modeling	The report includes appropriately applied models for regression, classification, and/or clustering. The method and results of the modeling are clearly described, as are the limitations of the analysis.
Written communication	The report is clearly written, with written analysis that synthesizes and explains the visualization and modeling aspects of the report, and at an appropriate level of technicality.
Clarity and appropriateness of conclusions	The report uses the data to give a great deal of insight related to the chosen policy question. Uncertainty and limitations of the data and analyses in the report are clearly communicated. The report includes enough information about the context of the policy question so that a non-expert can clearly understand the issue and its importance.
Notebook(s) with code	Along with the report, student submits well-commented notebook(s) with Python code used for the project.
Presentation	Student gives a clear and engaging presentation that highlights the main results of the project, as well as any major challenges or limitations. Presentation includes descriptive title, one appropriate and compelling data visualization, and several key takeaways.

Fig. 4: *Rubric for the policy project that formed a core component of the formal assessment of students throughout the course.*

[Buo18] Buolamwini, Joy, and Timnit Gebru. *Gender shades: Intersectional accuracy disparities in commercial gender classification*. Conference on fairness, accountability and transparency. PMLR, 2018. http://proceedings.mlr.press/v81/buolamwini18a.html

[Cai19] Cairo, Alberto. *How charts lie: Getting smarter about visual information*. WW Norton & Company, 2019.

[Dev17] De Veaux, Richard D., Mahesh Agarwal, Maia Averett, Benjamin S. Baumer, Andrew Bray, Thomas C. Bressoud, Lance Bryant et al. *Curriculum guidelines for undergraduate programs in data science*. Annual Review of Statistics and Its Application 4 (2017): 15-30. https://doi.org/10.1146/annurev-statistics-060116-053930

[Dig20] D'Ignazio, Catherine, and Lauren F. Klein. *Data Feminism*. MIT press, 2020.

[Eng01] Engeström, Yrjö. *Expansive learning at work: Toward an activity theoretical reconceptualization*. Journal of education and work 14, no. 1 (2001): 133-156. https://doi.org/10.1080/13639080020028747

[Hun07] Hunter, J.D., *Matplotlib: A 2D Graphics Environment*. Computing in Science & Engineering, vol. 9, no. 3 (2007): 90-95. https://doi.org/10.1109/MCSE.2007.55

[Jor21] Jordahl, Kelsey et al. 2021. *Geopandas/geopandas: V0.10.2*. Zenodo. https://doi.org/10.5281/zenodo.5573592.

[Mck10] McKinney, Wes. *Data structures for statistical computing in python*. In Proceedings of the 9th Python in Science Conference, vol. 445, no. 1, pp. 51-56. 2010. https://doi.org/10.25080/Majora-92bf1922-00a

[NAS18] National Academies of Sciences, Engineering, and Medicine. *Data science for undergraduates: Opportunities and options*. National Academies Press, 2018.

[One16] O'Neil, Cathy. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway Books, 2016.

[Ped11] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. *Scikit-learn: Machine learning in Python*. the Journal of machine Learning research 12 (2011): 2825-2830. https://dl.acm.org/doi/10.5555/1953048.2078195

[Pen16] Penuel, William R., Daniela K. DiGiacomo, Katie Van Horne, and Ben Kirshner. *A Social Practice Theory of Learning and Becoming across Contexts and Time*. Frontline Learning Research 4, no. 4 (2016): 30-38. http://dx.doi.org/10.14786/flr.v4i4.205

[Pro22] Project Jupyter, 2022. *jupyterlab/jupyterlab: JupyterLab 3.4.3* https://github.com/jupyterlab/jupyterlab

[The22] The Pandas Development Team, 2022. *pandas-dev/pandas: Pandas 1.4.2*. Zenodo. https://doi.org/10.5281/zenodo.6408044

[Was21] Waskom, Michael L. *Seaborn: statistical data visualization*. Journal of Open Source Software 6, no. 60 (2021): 3021. https://doi.org/10.21105/joss.03021

[Wil19] Wilke, Claus O. *Fundamentals of data visualization: a primer on making informative and compelling figures*. O'Reilly Media, 2019.

Papyri: better documentation for the scientific ecosystem in Jupyter

Matthias Bussonnier[‡§*], Camille Carvalho[¶‖]

Abstract—We present here the idea behind Papyri, a framework we are developing to provide a better documentation experience for the scientific ecosystem. In particular, we wish to provide a documentation browser (from within Jupyter or other IDEs and Python editors) that gives a unified experience, cross library navigation search and indexing. By decoupling documentation generation from rendering we hope this can help address some of the documentation accessibility concerns, and allow customisation based on users' preferences.

Index Terms—Documentation, Jupyter, ecosystem, accessibility

Introduction

Over the past decades, the *Python* ecosystem has grown rapidly, and one of the last bastion where some of the proprietary competition tools shine is integrated documentation. Indeed, open-source libraries are usually developed in distributed settings that can make it hard to develop coherent and integrated systems.

While a number of tools and documentations exists (and improvements are made everyday), most efforts attempt to build documentation in an isolated way, inherently creating a heterogeneous framework. The consequences are twofolds: (i) it becomes difficult for newcomers to grasp the tools properly, (ii) there is a lack of cohesion and of unified framework due to library authors making their proper choices as well as having to maintain build scripts or services.

Many users, colleagues, and members of the community have been frustrated with the documentation experience in the Python ecosystem. Given a library, who hasn't struggled to find the "official" website for the documentation ? Often, users stumble across an old documentation version that is better ranked in their favorite search engine, and this impacts significantly the learning process of less experienced users.

On users' local machine, this process is affected by limited documentation rendering. Indeed, while in many Integrated Development Environments (IDEs) the inspector provides some documentation, users do not get access to the narrative, or the full documentation gallery. For Command Line Interface (CLI) users, documentation is often displayed as raw source where no navigation is possible. On the maintainers' side, the final documentation rendering is less of a priority. Rather, maintainers should aim at making users gain from improvement in the rendering without having to rebuild all the docs.

Conda-Forge [CFRG] has shown that concerted efforts can give a much better experience to end-users, and in today's world where it is ubiquitous to share libraries source on code platforms, perform continuous integration and many other tools, we believe a better documentation framework for many of the libraries of the scientific Python should be available.

Thus, against all advice we received and based on our own experience, we have decided to rebuild an *opinionated* documentation framework, from scratch, and with minimal dependencies: *Papyri*. Papyri focuses on building an intermediate documentation representation format, that lets us decouple building, and rendering the docs. This highly simplifies many operations and gives us access to many desired features that were not available up to now.

In what follows, we provide the framework in which Papyri has been created and present its objectives (context and goals), we describe the Papyri features (format, installation, and usage), then present its current implementation. We end this paper with comments on current challenges and future work.

Context and objectives

Through out the paper, we will draw several comparisons between documentation building and compiled languages. Also, we will borrow and adapt commonly used terminology. In particular, similarities with "ahead-of-time" (AOT) [AOT], "just-in-time"" (JIT) [JIT], intermediate representation (IR) [IR], link-time optimization (LTO) [LTO], static vs dynamic linking will be highlighted. This allows us to clarify the presentation of the underlying architecture. However, there is no requirement to be familiar with the above to understand the concepts underneath Papyri. In that context, we wish to discuss documentation building as a process from a source-code meant for a machine to a final output targeting the flesh and blood machine between the keyboard and the chair.

Current tools and limitations

In the scientific Python ecosystem, it is well known that *Docutils* [docutils] and *Sphinx* [sphinx] are major cornerstones for publishing HTML documentation for Python. In fact, they are used by all the libraries in this ecosystem. While a few alternatives exist, most tools and services have some internal knowledge of Sphinx. For instance, Read the Docs [RTD] provides a specific

* *Corresponding author: bussonniermatthias@gmail.com*
‡ *QuanSight, Inc*
§ *Digital Ours Lab, SARL.*
¶ *University of California Merced, Merced, CA, USA*
‖ *Univ Lyon, INSA Lyon, UJM, UCBL, ECL, CNRS UMR 5208, ICJ, F-69621, France*

Sphinx theme [RTD-theme] users can opt-in to, *Jupyter-book* [JPYBOOK] is built on top of Sphinx, and *MyST* parser [MYST] (which is made to allow markdown in documentation) targets Sphinx as a backend, to name a few. All of the above provide an "ahead-of-time" documentation compilation and rendering, which is slow and computationally intensive. When a project needs its specific plugins, extensions and configurations to properly build (which is almost always the case), it is relatively difficult to build documentation for a single object (like a single function, module or class). This makes AOT tools difficult to use for interactive exploration. One can then consider a JIT approach, as done for *Docrepr* [DOCREPR] (integrated both in *Jupyter* and *Spyder* [Spyder]). However in that case, interactive documentation lacks inline plots, crosslinks, indexing, search and many custom directives.

Some of the above limitations are inherent to the design of documentation build tools that were intended for a separate documentation construction. While Sphinx does provide features like *intersphinx*, link resolutions are done at the documentation building phase. Thus, this is inherently unidirectional, and can break easily. To illustrate this, we consider *NumPy* [NP] and *SciPy* [SP], two extremely close libraries. In order to obtain proper cross-linked documentation, one is required to perform at least five steps:

- build NumPy documentation
- publish NumPy `object.inv` file.
- (re)build SciPy documentation using NumPy `obj.inv` file.
- publish SciPy `object.inv` file
- (re)build NumPy docs to make use of SciPy's `obj.inv`

Only then can both SciPy's and NumPy's documentation refer to each other. As one can expect, cross links break every time a new version of a library is published[1]. Pre-produced HTML in IDEs and other tools are then prone to error and difficult to maintain. This also raises security issues: some institutions become reluctant to use tools like Docrepr or viewing pre-produced HTML.

Docstrings format

The *Numpydoc* format is ubiquitous among the scientific ecosystem [NPDOC]. It is loosely based on *reStructuredText (RST)* syntax, and despite supporting full RST syntax, docstrings rarely contain full-featured directive. Maintainers are confronted to the following dilemma:

- keep the docstrings simple. This means mostly text-based docstrings with few directive for efficient readability. The end-user may be exposed to raw docstring, there is no on-the-fly directive interpretation. This is the case for tools such as *IPython* and Jupyter.
- write an extensive docstring. This includes references, and directive that potentially creates graphics, tables and more, allowing an enriched end-user experience. However this may be computationally intensive, and executing code to view docs could be a security risk.

Other factors impact this choice: (i) users, (ii) format, (iii) runtime. IDE users or non-Terminal users motivate to push for extensive docstrings. Tools like Docrepr can mitigate this problem by allowing partial rendering. However, users are often exposed to raw docstrings (see for example the *SymPy* discussion[2] on how equations should be displayed in docstrings, and left panel of Figure 1). In terms of format, markdown is appealing, however inconsistencies in the rendering will be created between libraries. Finally, some libraries can dynamically modify their docstring at runtime. While this sometime avoids using directives, it ends up being more expensive (runtime costs, complex maintenance, and contribution costs).

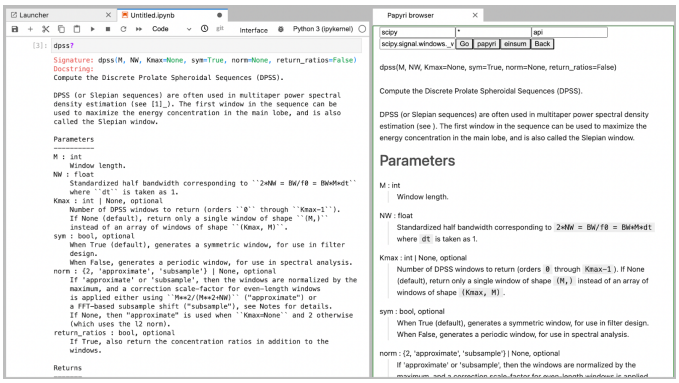


Fig. 1: *The following screenshot shows the help for* `scipy.signal.dpss`*, as currently accessible (left), as shown by Papyri for Jupyterlab extension (right). An extended version of the right pannel is displayed in Figure 4.*

Objectives of the project

We now layout the objectives of the Papyri documentation framework. Let us emphasize that the project is in no way intended to replace or cover many features included in well-established documentation tools such as Sphinx or Jupyter-book. Those projects are extremely flexible and meet the needs of their users for publishing a standalone documentation website of PDFs. The Papyri project addresses specific documentation challenges (mentioned above), we present below what is (and what is not) the scope of work.

Goal (a): design a non-generic (non fully customisable) website builder. When authors want or need complete control of the output and wide personalisation options, or branding, then Papyri is not likely the project to look at. That is to say single-project websites where appearance, layout, domain need to be controlled by the author is not part of the objectives.

Goal (b): create a uniform documentation structure and syntax. The Papyri project prescribes stricter requirements in terms of format, structure, and syntax compared to other tools such as Docutils and Sphinx. When possible, the documentation follows the *Diátaxis* Framework [DT]. This provides a uniform documentation setup and syntax, simplifying contributions to the project and easing error catching at compile time. Such strict environment is qualitatively supported by a number of documentation fixes done upstream during the development stage of the project[3]. Since Papyri is not fully customisable, users who are already using documentation tools such as Sphinx, *mkdocs* [mkdocs] and others should expect their project to require minor modifications to work with Papyri.

Goal (c): provide accessibility and user proficiency. Accessibility is a top priority of the project. To that aim, items are associated to semantic meaning as much as possible, and

1. ipython/ipython#12210, numpy/numpy#21016, & #29073

2. sympy/sympy#14963

3. Tests have been performed on NumPy, SciPy.

documentation rendering is separated from documentation building phase. That way, accessibility features such as high contract themes (for better text-to-speech (TTS) raw data), early example highlights (for newcomers) and type annotation (for advanced users) can be quickly available. With the uniform documentation structure, this provides a coherent experience where users become more comfortable finding information in a single location (see Figure 1).

Goal (d): make documentation building simple, fast, and independent. One objective of the project is to make documentation installation and rendering relatively straightforward and fast. To that aim, the project includes relative independence of documentation building across libraries, allowing bidirectional cross links (i.e. both forward and backward links between pages) to be maintained more easily. In other words, a single library can be built without the need to access documentation from another. Also, the project should include straightforward lookup documentation for an object from the interactive read–eval–print loop (REPL). Finally, efforts are put to limit the installation speed (to avoid polynomial growth when installing packages on large distributed systems).

The Papyri solution

In this section we describe in more detail how Papyri has been implemented to address the objectives mentioned above.

Making documentation a multi-step process

When using current documentation tools, customisation made by maintainers usually falls into the following two categories:

- simpler input convenience,
- modification of final rendering.

This first category often requires arbitrary code execution and must import the library currently being built. This is the case for example for the use of `.. code-block:::`, or custom `:rc:` directive. The second one offers a more user friendly environment. For example, *sphinx-copybutton* [sphinx-copybutton] adds a button to easily copy code snippets in a single click, and *pydata-sphinx-theme* [pydata-sphinx-theme] or *sphinx-rtd-dark-mode* provide a different appearance. As a consequence, developers must make choices on behalf of their end-users: this may concern syntax highlights, type annotations display, light/dark theme.

Being able to modify extensions and re-render the documentation without the rebuilding and executing stage is quite appealing. Thus, the building phase in Papyri (collecting documentation information) is separated from the rendering phase (Objective (c)): at this step, Papyri has no knowledge and no configuration options that permit to modify the appearance of the final documentation. Additionally, the optional rendering process has no knowledge of the building step, and can be run without accessing the libraries involved.

This kind of technique is commonly used in the field of compilers with the usage of Single Compilation Unit [SCU] and Intermediate Representation [IR], but to our knowledge, it has not been implemented for documentation in the Python ecosystem. As mentioned before, this separation is key to achieving many features proposed in Objectives (c), (d) (see Figure 2).

Intermediate Representation for Documentation (IRD)

IRD format: Papyri relies on standard interchangeable "Intermediate Representation for Documentation" (IRD) format. This allows to reduce operation complexity of the documentation build. For example, given M documentation producers and N renderers, a full documentation build would be O(MN) (each renderer needs to understand each producer). If each producer only cares about producing IRD, and if each renderer only consumes it, then one can reduce to O(M+N). Additionally, one can take IRD from multiple producers at once, and render them all to a single target, breaking the silos between libraries.

At the moment, IRD files are currently separated into four main categories roughly following the Diátaxis framework [DT] and some technical needs:

- API files describe the documentation for a single object, expressed as a *JSON* object. When possible, the information is encoded semantically (Objective (c)). Files are organized based on the fully-qualified name of the Python object they reference, and contain either absolute reference to another object (library, version and identifier), or delayed references to objects that may exist in another library. Some extra per-object meta information like file/line number of definitions can be stored as well.
- Narrative files are similar to API files, except that they do not represent a given object, but possess a previous/next page. They are organised in an ordered tree related to the table of content.
- Example files are a non-ordered collection of files.
- Assets files are untouched binary resource archive files that can be referenced by any of the above three ones. They are the only ones that contain backward references, and no forward references.

In addition to the four categories above, metadata about the current package is stored: this includes library name, current version, *PyPi* name, *GitHub* repository slug[4], maintainers' names, logo, issue tracker and others. In particular, metadata allows us to auto-generate links to issue trackers, and to source files when rendering. In order to properly resolve some references and normalize links convention, we also store a mapping from fully qualified names to canonical ones.

Let us make some remarks about the current stage of IRD format. The exact structure of package metadata has not been defined yet. At the moment it is reduced to the minimum functionality. While formats such as *codemeta* [CODEMETA] could be adopted, in order to avoid information duplication we rely on metadata either present in the published packages already or extracted from Github repository sources. Also, IRD files must be standardized in order to achieve a uniform syntax structure (Objective (b)). In this paper, we do not discuss IRD files distribution. Last, the final specification of IRD files is still in progress and regularly undergoes major changes (even now). Thus, we invite contributors to consult the current state of implementation on the GitHub repository [Papyri]. Once the IRD format is more stable, this will be published as a JSON schema, with full specification and more in-depth description.

4. "slug" is the common term that refers to the various combinations of organization name/user name/repository name, that uniquely identifies a repository on a platform like GitHub.

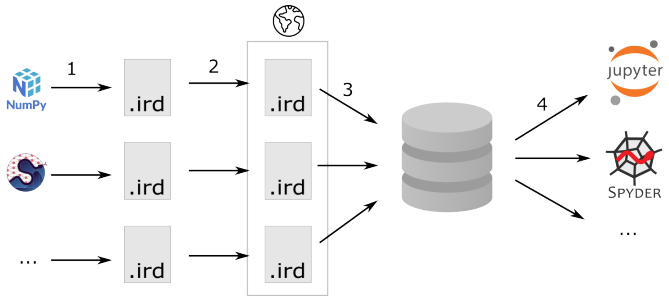


Fig. 2: *Sketch representing how to build documentation with Papyri. Step 1: Each project builds an IRD bundle that contains semantic information about the project documentation. Step 2: the IRD bundles are publihsed online. Step 3: users install IRD bundles locally on their machine, pages get corsslinked, indexed, etc. Step 4: IDEs render documentation on-the-fly, taking into consideration users' preferences.*

IRD bundles: Once a library has collected IRD representation for all documentation items (functions, class, narrative sections, tutorials, examples), Papyri consolidates them into what we will refer to as IRD bundles. A Bundle gathers all IRD files and metadata for a single version of a library[5]. Bundles are a convenient unit to speak about publication, installation, or update of a given library documentation files.

Unlike package installation, IRD bundles do not have the notion of dependencies. Thus, a fully fledged package manager is not necessary, and one can simply download corresponding files and unpack them at the installation phase.

Additionally, IRD bundles for multiple versions of the same library (or conflicting libraries) are not inherently problematic as they can be shared across multiple environments.

From a security standpoint, installing IRD bundles does not require the execution of arbitrary code. This is a critical element for adoption in deployments. There exists as well an opportunity to provide localized variants at the IRD installation time (IRD bundle translations haven't been explored exhaustively at the moment).

IRD and high level usage

Papyri-based documentation involves three broad categories of stakeholders (library maintainers, end-users, IDE developers), and processes. This leads to certain requirements for IRD files and bundles.

On the maintainers' side, the goal is to ensure that Papyri can build IRD files, and publish IRD bundles. Creation of IRD files and bundles is the most computationally intensive step. It may require complex dependencies, or specific plugins. Thus, this can be a multi-step process, or one can use external tooling (not related to Papyri nor using Python) to create them. Visual appearance and rendering of documentation is not taken into account in this process. Overall, building IRD files and bundles takes about the same amount of time as running a full Sphinx build. The limiting factor is often associated to executing library examples and code snippets. For example, building SciPy & NumPy documentation IRD files on a 2021 Macbook Pro M1 (base model), including executing examples in most docstrings and type inferring most examples (with most variables semantically inferred) can take several minutes.

End-users are responsible for installing desired IRD bundles. In most cases, it will consist of IRD bundles from already installed libraries. While Papyri is not currently integrated with package managers or IDEs, one could imagine this process being automatic, or on demand. This step should be fairly efficient as it mostly requires downloading and unpacking IRD files.

Finally, IDEs developers want to make sure IRD files can be properly rendered and browsed by their users when requested. This may potentially take into account users' preferences, and may provide added values such as indexing, searching, bookmarks and others, as seen in rustsdocs, devdocs.io.

Current implementation

We present here some of the technological choices made in the current Papyri implementation. At the moment, it is only targeting a subset of projects and users that could make use of IRD files and bundles. As a consequence, it is constrained in order to minimize the current scope and efforts development. Understanding the implementation is **not necessary to use Papyri** neither as a project maintainer nor as a user, but it can help understanding some of the current limitations.

Additionally, nothing prevents alternatives and complementary implementations with different choices: as long as other implementations can produce (or consume) IRD bundles, they should be perfectly compatible and work together.

The following sections are thus mostly informative to understand the state of the current code base. In particular we restricted ourselves to:

- Producing IRD bundles for the core scientific Python projects (Numpy, SciPy, Matplotlib...)
- Rendering IRD documentation for a single user on their local machine.

Finally, some of the technological choices have no other justification than the main developer having interests in them, or making iterations on IRD format and main code base faster.

IRD files generation

The current implementation of Papyri only targets some compatibility with Sphinx (a website and PDF documentation builder), reStructuredText (RST) as narrative documentation syntax and Numpydoc (both a project and standard for docstring formatting).

These are widely used by a majority of the core scientific Python ecosystem, and thus having Papyri and IRD bundles compatible with existing projects is critical. We estimate that about 85%-90% of current documentation pages being built with Sphinx, RST and Numpydoc can be built with Papyri. Future work includes extensions to be compatible with MyST (a project to bring markdown syntax to Sphinx), but this is not a priority.

5. One could have IRD bundles not attached to a particular library. For example, this can be done if an author wishes to provide only a set of examples or tutorials. We will not discuss this case further here.

To understand RST Syntax in narrative documentation, RST documents need to be parsed. To do so, Papyri uses tree-sitter [TS] and tree-sitter-rst [TSRST] projects, allowing us to extract an "Abstract Syntax Tree" (AST) from the text files. When using tree-sitter, AST nodes contain bytes-offsets into the original text buffer. Then one can easily "unparse" an AST node when necessary. This is relatively convenient for handling custom directives and edge cases (for instance, when projects rely on a loose definition of the RST syntax). Let us provide an example: RST directives are usually of the form:

```
.. directive:: arguments

    body
```

While technically there is no space before the `::`, Docutils and Sphinx will not create errors when building the documentation. Due to our choice of a rigid (but unified) structure, we use tree-sitter that indicates an error node if there is an extra space. This allows us to check for error nodes, unparse, add heuristics to restore a proper syntax, then parse again to obtain the new node.

Alternatively, a number of directives like `warnings`, `notes` `admonitions` still contain valid RST. Instead of storing the directive with the raw text, we parse the full document (potentially finding invalid syntax), and unparse to the raw text only if the directive requires it.

Serialisation of data structure into IRD files is currently using a custom serialiser. Future work includes maybe swapping to *msgspec* [msgspec]. The AST objects are completely typed, however they contain a number of unions and sequences of unions. It turns out, many frameworks like *pydantic* [pydantic] do not support sequences of unions where each item in the union may be of a different type. To our knowledge, there are just few other documentation related projects that treat AST as an intermediate object with a stable format that can be manipulated by external tools. In particular, the most popular one is Pandoc [pandoc], a project meant to convert from many document types to plenty of other ones.

The current Papyri strategy is to type-infer all code examples with *Jedi* [JEDI], and pre-syntax highlight using *pygments* when possible.

IRD File Installation

Download and installation of IRD files is done concurrently using *httpx* [httpx], with *Trio* [Trio] as an async framework, allowing us to download files concurrently.

The current implementation of Papyri targets Python documentation and is written in Python. We can then query the existing version of Python libraries installed, and infer the appropriate version of the requested documentation. At the moment, the implementation is set to tentatively guess relevant libraries versions when the exact version number is missing from the install command.

For convenience and performance, IRD bundles are being post-processed and stored in a different format. For local rendering, we mostly need to perform the following operations:

1) Query graph information about cross-links across documents.
2) Render a single page.
3) Access raw data (e.g. images).

We also assume that IRD files may be infrequently updated, that disk space is limited, and that installing or running services (like a database server) are not necessary available. This provides an adapted framework to test Papyri on an end-user machine.

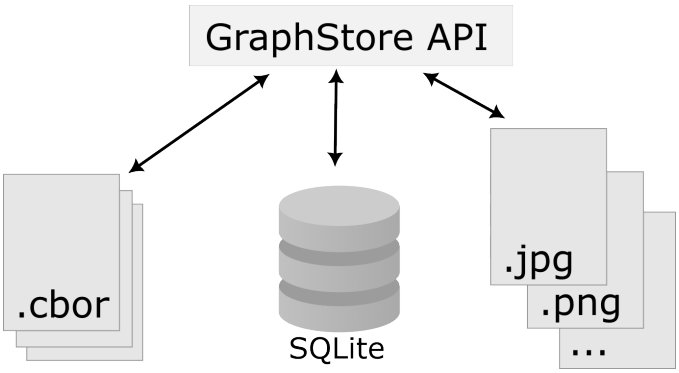


Fig. 3: *Sketch representing how Papyri stores information in 3 different format depending on access patterns: a SQLite database for relationship information, on-disk CBOR files for more compact storate of IRD, and RAW files (e.g. Images). A GraphStore API abstracts all access and takes care of maintinaing consistency.*

With those requirements we decided to use a combination of *SQLite* (an in-process database engine), *Concise Binary Object Representation (CBOR)* and raw storage to better reflect the access pattern (see Figure 3).

SQLite allows us to easily query for object existence, and graph information (relationship between objects) at runtime. It is optimized for infrequent reading access. Currently many queries are done at runtime, when rendering documentation. The goal is to move most of SQLite information resolving step at the installation time (such as looking for inter-libraries links) once the codebase and IRD format have stabilized. SQLite is less strongly typed than other relational or graph database and needs custom logic, but is ubiquitous on all systems and does not need a separate server process, making it an easy choice of database.

CBOR is a more space efficient alternative to JSON. In particular, keys in IRD are often highly redundant, and can be highly optimized when using CBOR. Storing IRD in CBOR thus reduces disk usage and can also allow faster deserialization without requiring potentially CPU intensive compression/decompression. This is a good compromise for potentially low performance users' machines.

Raw storage is used for binary blobs which need to be accessed without further processing. This typically refers to images, and raw storage can be accessed with standard tools like image viewers.

Finally, access to all of these resources is provided via an internal `GraphStore` API which is agnostic of the backend, but ensures consistency of operations like adding/removing/replacing documents. Figure 3 summarizes this process.

Of course the above choices depend on the context where documentation is rendered and viewed. For example, an online archive intended to browse documentation for multiple projects and versions may decide to use an actual graph database for object relationship, and store other files on a Content Delivery Network or blob storage for random access.

Documentation Rendering

The current Papyri implementation includes a certain number of rendering engines (presented below). Each of them mostly consists of fetching a single page with its metadata, and walking

through the IRD AST tree, and rendering each node with users' preferences.

- An ASCII terminal renders using *Jinja2* [Jinja2]. This can be useful for piping documentation to other tools like `grep`, `less`, `cat`. Then one can work in a highly restricted environment, making sure that reading the documentation is coherent. This can serve as a proxy for screen reading.
- A Textual User Interface browser renders using *urwid*. Navigation within the terminal is possible, one can reflow long lines on resized windows, and even open image files in external editors. Nonetheless, several bugs have been encountered in urwid. The project aims at replacing the CLI IPython *question mark operator* (`obj?`) interface (which currently only shows raw docstrings) in urwid with a new one written with *Rich*/*Textual*. For this interface, having images stored raw on disk is useful as it allows us to directly call into a system image viewer to display them.
- A JIT rendering engine uses Jinja2, *Quart* [quart], Trio. Quart is an async version of *flask* [flask]. This option contains the most features, and therefore is the main one used for development. This environment lets us iterate over the rendering engine rapidly. When exploring the User Interface design and navigation, we found that a list of back references has limited uses. Indeed, it is can be challenging to judge the relevance of back references, as well as their relationship to each other. By playing with a network graph visualisation (see Figure 5)), we can identify clusters of similar information within back references. Of course, this identification has limits especially when pages have a large number of back references (where the graph becomes too busy). This illustrate as well a strength of the Papyri architecture: creating this network visualization did not require any regeneration of the documentation, one simply updates the template and re-renders the current page as needed.
- A static AOT rendering of all the existing pages that can be rendered ahead of time uses the same class as the JIT rendering. Basically, this loops through all entries in the SQLite database and renders each item independently. This renderer is mostly used for exhaustive testing and performance measures for Papyri. This can render most of the API documentation of IPython, *Astropy* [astropy], *Dask* and *distributed* [Dask], *Matplotlib* [MPL], [MPL-DOI], *Networkx* [NX], NumPy [NP], *Pandas*, Papyri, SciPy, *Scikit-image* and others. It can represent ~28000 pages in ~60 seconds (that is ~450 pages/s on a recent Macbook pro M1).

For all of the above renderers, profiling shows that documentation rendering is mostly limited by object de-serialisation from disk and Jinja2 templating engine. In the early project development phase, we attempted to write a static HTML renderer in a compiled language (like Rust, using compiled and typed checked templates). This provided a speedup of roughly a factor 10. However, its implementation is now out of sync with the main Papyri code base.

Finally, a JupyterLab extension is currently in progress. The documentation then presents itself as a side-panel and is capable of basic browsing and rendering (see Figure 1 and Figure 4). The model uses *typescript*, react and native JupyterLab component. Future goals include improving/replacing the JupyterLab's question mark operator (`obj?`) and the JupyterLab Inspector (when possible). A screenshot of the current development version of the JupyterLab extension can be seen in Figure 4.

Challenges

We mentioned above some limitations we encountered (in rendering usage for instance) and what will be done in the future to address them. We provide below some limitations related to syntax choices, and broader opportunities that arise from the Papyri project.

Limitations

The decoupling of the building and rendering phases is key in Papyri. However, it requires us to come up with a method that uniquely identifies each object. In particular, this is essential in order to link any object documentation without accessing the IRD bundles build from all the libraries. To that aim, we use the fully qualified names of an object. Namely, each object is identified by the concatenation of the module in which it is defined, with its local name. Nonetheless, several particular cases need specific treatment.

- To mirror the Python syntax, is it easy to use `.` to concatenate both parts. Unfortunately, that leads to some ambiguity when modules re-export functions have the same name. For example, if one types

  ```
  # module mylib/__init__.py

  from .mything import mything
  ```

  then `mylib.mything` is ambiguous both with respect to the `mything` submodule, and the reexported object. In future versions, the chosen convention will use `:` as a module/name separator.
- Decorated functions or other dynamic approaches to expose functions to users end up having `<local>>` in their fully qualified names, which is invalid.
- Many built-in functions (`np.sin`, `np.cos`, etc.) do not have a fully qualified name that can be extracted by object introspection. We believe it should be possible to identify those via other means like docstring hash (to be explored).
- Fully qualified names are often not canonical names (i.e. the name typically used for import). While we made efforts to create a mapping from one to another, finding the canonical name automatically is not always straightforward.
- There are also challenges with case sensitivity. For example for *MacOS* file systems, a couple of objects may unfortunately refer to the same IRD file on disk. To address this, a case-sensitive hash is appended at the end of the filename.
- Many libraries have a syntax that *looks* right once rendered to HTML while not following proper syntax, or a syntax that relies on specificities of Docutils and Sphinx rendering/parsing.
- Many custom directive plugins cannot be reused from Sphinx. These will need to be reimplemented.

Future possibilities

Beyond what has been presented in this paper, there are several opportunities to improve and extend what Papyri can allow for the scientific Python ecosystem.

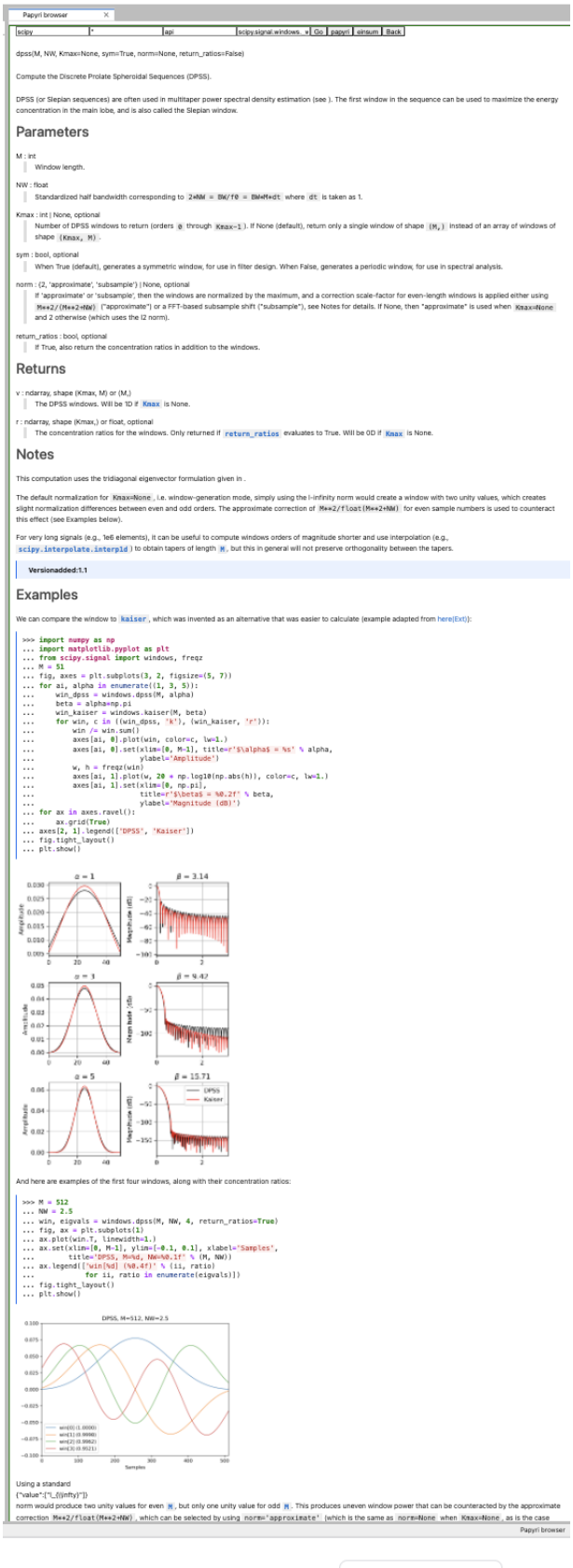


Fig. 4: *Example of extended view of the Papyri documentation for Jupyterlab extension (here for SciPy). Code examples can now include plots. Most token in each examples are linked to the corresponding page. Early navigation bar is visible at the top.*

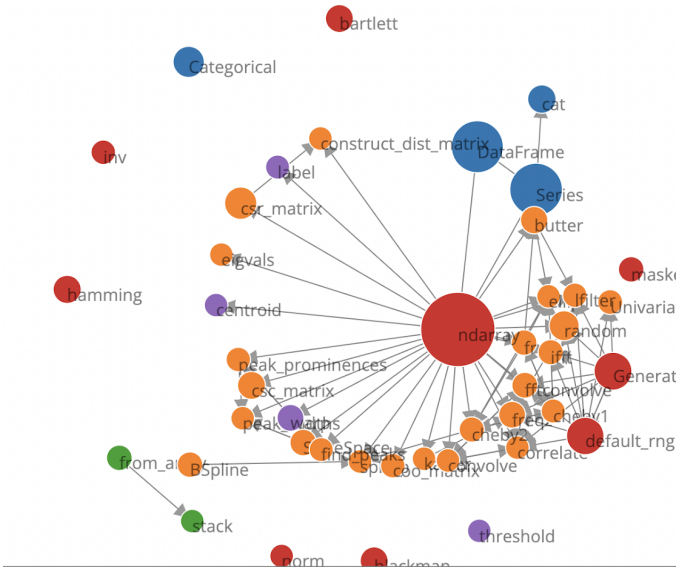


Fig. 5: *Local graph (made with D3.js [D3js]) representing the connections among the most important nodes around current page across many libraries, when viewing* `numpy.ndarray`*. Nodes are sized with respect to the number of incomming links, and colored with respect to their library. This graph is generated at rendering time, and is updated depending on the libraries currently installed. This graph helps identify related functions and documentation. It can become challenging to read for highly connected items as seen here for* `numpy.ndarray`*.*

The first area is the ability to build IRD bundles on Continuous Integration platforms. Services like GitHub action, Azure pipeline and many others are already setup to test packages. We hope to leverage this infrastructure to build IRD files and make them available to users.

A second area is hosting of intermediate IRD files. While the current prototype is hosted by http index using GitHub pages, it is likely not a sustainable hosting platform as disk space is limited. To our knowledge, IRD files are smaller in size than HTML documentation, we hope that other platforms like Read the Docs can be leveraged. This could provide a single domain that renders the documentation for multiple libraries, thus avoiding the display of many library subdomains. This contributes to giving a more unified experience for users.

It should be possible for projects to avoid using many dynamic docstrings interpolation that are used to document `*args` and `**kwargs`. This would make sources easier to read, and potentially have some speedup at the library import time.

Once a (given and appropriately used by its users) library uses an IDE that supports Papyri for documentation, docstring syntax could be exchanged for markdown.

As IRD files are structured, it should be feasible to provide cross-version information in the documentation. For example, if one installs multiple versions of IRD bundles for a library, then assuming the user does not use the latest version, the renderer could inspect IRD files from previous/future versions to indicate the range of versions for which the documentation has not changed. Upon additional efforts, it should be possible to infer *when* a parameter was removed, or will be removed, or to simply display the difference between two versions.

Conclusion

To address some of the current limitations in documentation accessibility, building and maintaining, we have provided a new documentation framework called Papyri. We presented its features and underlying implementation choices (such as crosslink maintenance, decoupling building and rendering phases, enriching the rendering features, using the IRD format to create a unified syntax structure, etc.). While the project is still at its early stage, clear impacts can already be seen on the availability of high-quality documentation for end-users, and on the workload reduction for maintainers. Building IRD format opened a wide range of technical possibilities, and contributes to improving users' experience (and therefore the success of the scientific Python ecosystem). This may become necessary for users to navigate in an exponentially growing ecosystem.

Acknowledgments

The authors want to thank S. Gallegos (author of tree-sitter-rst), J. L. Cano Rodríguez and E. Holscher (Read The Docs), C. Holdgraf (2i2c), B. Granger and F. Pérez (Jupyter Project), T. Allard and I. Presedo-Floyd (QuanSight) for their useful feedback and help on this project.

Funding

M. B. received a 2-year grant from the Chan Zuckerberg Initiative (CZI) Essential Open Source Software for Science (EOS) – EOSS4-0000000017 via the NumFOCUS 501(3)c non profit to develop the Papyri project.

References

[AOT] https://en.wikipedia.org/wiki/Ahead-of-time_compilation

[CFRG] conda-forge community. (2015). The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem. Zenodo. http://doi.org/10.5281/zenodo.4774216

[CODEMETA] https://codemeta.github.io/

[D3js] https://d3js.org/

[DOCREPR] https://github.com/spyder-ide/docrepr

[DT] https://diataxis.fr/

[Dask] Dask Development Team (2016). Dask: Library for dynamic task scheduling, https://dask.org

[IR] https://en.wikipedia.org/wiki/Intermediate_representation

[JEDI] https://github.com/davidhalter/jedi

[JIT] https://en.wikipedia.org/wiki/Just-in-time_compilation

[JPYBOOK] https://jupyterbook.org/

[Jinja2] https://jinja.palletsprojects.com/

[LTO] https://en.wikipedia.org/wiki/Interprocedural_optimization

[MPL-DOI] https://doi.org/10.5281/zenodo.6513224

[MPL] J.D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007,

[MYST] https://myst-parser.readthedocs.io/en/latest/

[NPDOC] https://numpydoc.readthedocs.io/en/latest/format.html

[NP] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

[NX] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[Papyri] https://github.com/jupyter/papyri

[RTD-theme] https://sphinx-rtd-theme.readthedocs.io/en/stable/

[RTD] https://readthedocs.org/

[SCU] https://en.wikipedia.org/wiki/Single_Compilation_Unit

[SP] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), 261-272. 10.1038/s41592-019-0686-2

[Spyder] https://www.spyder-ide.org/

[TSRST] https://github.com/stsewd/tree-sitter-rst

[TS] https://tree-sitter.github.io/tree-sitter/

[astropy] The Astropy Project: Building an inclusive, open-science project and status of the v2.0 core package, https://doi.org/10.48550/arXiv.1801.02634

[docutils] https://docutils.sourceforge.io/

[flask] https://flask.palletsprojects.com/en/2.1.x/

[httpx] https://www.python-httpx.org/

[mkdocs] https://www.mkdocs.org/

[msgspec] https://pypi.org/project/msgspec

[pandoc] https://pandoc.org/

[pydantic] https://pydantic-docs.helpmanual.io/

[pydata-sphinx-theme] https://pydata-sphinx-theme.readthedocs.io/en/stable/

[quart] https://pgjones.gitlab.io/quart/

[sphinx-copybutton] https://sphinx-copybutton.readthedocs.io/en/latest/

[sphinx] https://www.sphinx-doc.org/en/master/

[Trio] https://trio.readthedocs.io/

Bayesian Estimation and Forecasting of Time Series in statsmodels

Chad Fulton[‡*]

Abstract—`Statsmodels`, a Python library for statistical and econometric analysis, has traditionally focused on frequentist inference, including in its models for time series data. This paper introduces the powerful features for Bayesian inference of time series models that exist in `statsmodels`, with applications to model fitting, forecasting, time series decomposition, data simulation, and impulse response functions.

Index Terms—time series, forecasting, bayesian inference, Markov chain Monte Carlo, statsmodels

Introduction

`Statsmodels` [SP10] is a well-established Python library for statistical and econometric analysis, with support for a wide range of important model classes, including linear regression, ANOVA, generalized linear models (GLM), generalized additive models (GAM), mixed effects models, and time series models, among many others. In most cases, model fitting proceeds by using frequentist inference, such as maximum likelihood estimation (MLE). In this paper, we focus on the class of time series models [MPS11], support for which has grown substantially in `statsmodels` over the last decade. After introducing several of the most important new model classes – which are by default fitted using MLE – and their features – which include forecasting, time series decomposition and seasonal adjustment, data simulation, and impulse response analysis – we describe the powerful functions that enable users to apply Bayesian methods to a wide range of time series models.

Support for Bayesian inference in Python outside of `statsmodels` has also grown tremendously, particularly in the realm of probabilistic programming, and includes powerful libraries such as `PyMC3` [SWF16], `PyStan` [CGH+17], and `TensorFlow Probability` [DLT+17]. Meanwhile, `ArviZ` [KCHM19] provides many excellent tools for associated diagnostics and vizualisations. The aim of these libraries is to provide support for Bayesian analysis of a large class of models, and they make available both advanced techniques, including autotuning algorithms, and flexible model specification. By contrast, here we focus on simpler techniques. However, while the libraries above do include some support for time series models, this has not been their primary focus. As a result, introducing Bayesian inference for the well-developed stable of time series models in `statsmodels`, and providing access to the rich associated feature set already mentioned, presents a complementary option to these more general-purpose libraries.[1]

* *Corresponding author: chad.t.fulton@frb.gov*
‡ *Federal Reserve Board of Governors*

Time series analysis in `statsmodels`

A time series is a sequence of observations ordered in time, and time series data appear commonly in statistics, economics, finance, climate science, control systems, and signal processing, among many other fields. One distinguishing characteristic of many time series is that observations that are close in time tend to be more correlated, a feature known as autocorrelation. While successful analyses of time series data must account for this, statistical models can harness it to decompose a time series into trend, seasonal, and cyclical components, produce forecasts of future data, and study the propagation of shocks over time.

We now briefly review the models for time series data that are available in `statsmodels` and describe their features.[2]

Exponential smoothing models

Exponential smoothing models are constructed by combining one or more simple equations that each describe some aspect of the evolution of univariate time series data. While originally somewhat *ad hoc*, these models can be defined in terms of a proper statistical model (for example, see [HKOS08]). They have enjoyed considerable popularity in forecasting (for example, see the implementation in R described by [HA18]). A prototypical example that allows for trending data and a seasonal component – often known as the additive "Holt-Winters' method" – can be written as

$$\begin{aligned} l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}) \\ b_t &= \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \\ s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m} \end{aligned}$$

where l_t is the level of the series, b_t is the trend, s_t is the seasonal component of period m, and α, β, γ are parameters of the model. When augmented with an error term with some given probability distribution (usually Gaussian), likelihood-based inference can be used to estimate the parameters. In `statsmodels`,

1. In addition, it is possible to combine the sampling algorithms of PyMC3 with the time series models of `statsmodels`, although we will not discuss this approach in detail here. See, for example, https://www.statsmodels.org/v0.13.0/examples/notebooks/generated/statespace_sarimax_pymc3.html.

2. In addition to statistical models, `statsmodels` also provides a number of tools for exploratory data analysis, diagnostics, and hypothesis testing related to time series data; see https://www.statsmodels.org/stable/tsa.html.

additive exponential smoothing models can be constructed using the `statespace.ExponentialSmoothing` class.[3] The following code shows how to apply the additive Holt-Winters model above to model quarterly data on consumer prices:

```
import statsmodels.api as sm
# Load data
mdata = sm.datasets.macrodata.load().data
# Compute annualized consumer price inflation
y = np.log(mdata['cpi']).diff().iloc[1:] * 400

# Construct the Holt-Winters model
model_hw = sm.tsa.statespace.ExponentialSmoothing(
    y, trend=True, seasonal=12)
```

Structural time series models

Structural time series models, introduced by [Har90] and also sometimes known as unobserved components models, similarly decompose a univariate time series into trend, seasonal, cyclical, and irregular components:

$$y_t = \mu_t + \gamma_t + c_t + \varepsilon_t$$

where μ_t is the trend, γ_t is the seasonal component, c_t is the cyclical component, and $\varepsilon_t \sim N(0, \sigma^2)$ is the error term. However, this equation can be augmented in many ways, for example to include explanatory variables or an autoregressive component. In addition, there are many possible specifications for the trend, seasonal, and cyclical components, so that a wide variety of time series characteristics can be accommodated. In `statsmodels`, these models can be constructed from the `UnobservedComponents` class; a few examples are given in the following code:

```
# "Local level" model
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')
# "Local linear trend", with seasonal component
model_arma11 = sm.tsa.UnobservedComponents(
    y, 'lltrend', seasonal=4)
```

These models have become popular for time series analysis and forecasting, as they are flexible and the estimated components are intuitive. Indeed, Google's Causal Impact library [BGK+15] uses a Bayesian structural time series approach directly, and Facebook's Prophet library [TL17] uses a conceptually similar framework and is estimated using PyStan.

Autoregressive moving-average models

Autoregressive moving-average (ARMA) models, ubiquitous in time series applications, are well-supported in `statsmodels`, including their generalizations, abbreviated as "SARIMAX", that allow for integrated time series data, explanatory variables, and seasonal effects.[4] A general version of this model, excluding integration, can be written as

$$\begin{aligned} y_t &= x_t\beta + \xi_t \\ \xi_t &= \phi_1\xi_{t-1} + \cdots + \phi_p\xi_{t-p} + \varepsilon_t + \theta_1\varepsilon_{t-1} + \cdots + \theta_q\varepsilon_{t-q} \end{aligned}$$

where $\varepsilon_t \sim N(0, \sigma^2)$. These are constructed in `statsmodels` with the `ARIMA` class; the following code shows how to construct a variety of autoregressive moving-average models for consumer price data:

```
# AR(2) model
model_ar2 = sm.tsa.ARIMA(y, order=(2, 0, 0))

# ARMA(1, 1) model with explanatory variable
X = mdata['realint']
model_arma11 = sm.tsa.ARIMA(
    y, order=(1, 0, 1), exog=X)
# SARIMAX(p, d, q)x(P, D, Q, s) model
model_sarimax = sm.tsa.ARIMA(
    y, order=(p, d, q), seasonal_order=(P, D, Q, s))
```

While this class of models often produces highly competitive forecasts, it does not produce a decomposition of a time series into, for example, trend and seasonal components.

Vector autoregressive models

While the SARIMAX models above handle univariate series, `statsmodels` also has support for the multivariate generalization to vector autoregressive (VAR) models.[5] These models are written

$$y_t = \nu + \Phi_1 y_{t-1} + \cdots + \Phi_p y_{t-p} + \varepsilon_t$$

where y_t is now considered as an $m \times 1$ vector. As a result, the intercept ν is also an $m \times 1$ vector, the coefficients Φ_i are each $m \times m$ matrices, and the error term is $\varepsilon_t \sim N(0_m, \Omega)$, with Ω an $m \times m$ matrix. These models can be constructed in `statsmodels` using the `VARMAX` class, as follows[6]

```
# Multivariate dataset
z = (np.log(mdata['realgdp', 'realcons', 'cpi'])
       .diff().iloc[1:])

# VAR(1) model
model_var = sm.tsa.VARMAX(z, order=(1, 0))
```

Dynamic factor models

`statsmodels` also supports a second model for multivariate time series: the dynamic factor model (DFM). These models, often used for dimension reduction, posit a few unobserved factors, with autoregressive dynamics, that are used to explain the variation in the observed dataset. In `statsmodels`, there are two model classes, `DynamicFactor`` and `DynamicFactorMQ`, that can fit versions of the DFM. Here we focus on the `DynamicFactor` class, for which the model can be written

$$\begin{aligned} y_t &= \Lambda f_t + \varepsilon_t \\ f_t &= \Phi_1 f_{t-1} + \cdots + \Phi_p f_{t-p} + \eta_t \end{aligned}$$

Here again, the observation is assumed to be $m \times 1$, but the factors are $k \times 1$, where it is possible that $k << m$. As before, we assume conformable coefficient matrices and Gaussian errors.

The following code shows how to construct a DFM in `statsmodels`

```
# DFM with 2 factors that evolve as a VAR(3)
model_dfm = sm.tsa.DynamicFactor(
    z, k_factors=2, factor_order=3)
```

Linear Gaussian state space models

In `statsmodels`, each of the model classes introduced above (`statespace.ExponentialSmoothing`, `UnobservedComponents`, `ARIMA`, `VARMAX`,

3. A second class, `ETSModel`, can also be used for both additive and multiplicative models, and can exhibit superior performance with maximum likelihood estimation. However, it lacks some of the features relevant for Bayesian inference discussed in this paper.

4. Note that in `statsmodels`, models with explanatory variables are in the form of "regression with SARIMA errors".

5. `statsmodels` also supports vector moving-average (VMA) models using the same model class as described here for the VAR case, but, for brevity, we do not explicitly discuss them here.

6. A second class, `VAR`, can also be used to fit VAR models, using least squares. However, it lacks some of the features relevant for Bayesian inference discussed in this paper.

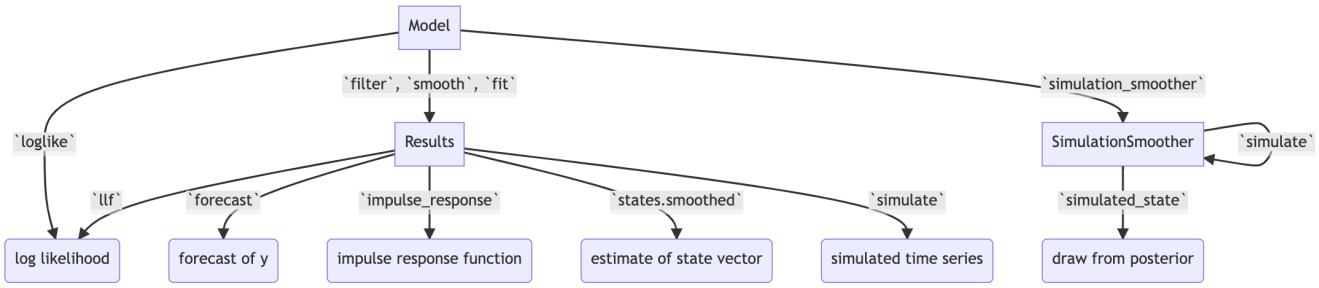


Fig. 1: *Selected functionality of state space models in statsmodels.*

`DynamicFactor`, and `DynamicFactorMQ`) are implemented as part of a broader class of models, referred to as linear Gaussian state space models (hereafter for brevity, simply "state space models" or SSM). This class of models can be written as

$$y_t = d_t + Z_t \alpha_t + \varepsilon_t \qquad \varepsilon_t \sim N(0, H_t)$$
$$\alpha_{t+1} = c_t + T_t \alpha_t + R_t \eta_t \qquad \eta_t \sim N(0, Q_t)$$

where α_t represents an unobserved vector containing the "state" of the dynamic system. In general, the model is multivariate, with y_t and ε_t $m \times 1$ vector, α_t $k \times 1$, and η_t *r times 1*.

Powerful tools exist for state space models to estimate the values of the unobserved state vector, compute the value of the likelihood function for frequentist inference, and perform posterior sampling for Bayesian inference. These tools include the celebrated Kalman filter and smoother and a simulation smoother, all of which are important for conducting Bayesian inference for these models.[7] The implementation in `statsmodels` largely follows the treatment in [DK12], and is described in more detail in [Ful15].

In addition to these key tools, state space models also admit general implementations of useful features such as forecasting, data simulation, time series decomposition, and impulse response analysis. As a consequence, each of these features extends to each of the time series models described above. Figure 1 presents a diagram showing how to produce these features, and the code below briefly introduces a subset of them.

```
# Construct the Model
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')

# Construct a simulation smoother
sim_ll = model_ll.simulation_smoother()

# Parameter values (variance of error and
# variance of level innovation, respectively)
params = [4, 0.75]

# Compute the log-likelihood of these parameters
llf = model_ll.loglike(params)

# `smooth` applies the Kalman filter and smoother
# with a given set of parameters and returns a
# Results object
results_ll = model_ll.smooth(params)

# Produce forecasts for the next 4 periods
fcast = results_ll.forecast(4)

# Produce a draw from the posterior distribution
# of the state vector
sim_ll.simulate()
draw = sim_ll.simulated_state
```

Nearly identical code could be used for any of the model classes introduced above, since they are all implemented as part of the same state space model framework. In the next section, we show how these features can be used to perform Bayesian inference with these models.

Bayesian inference via Markov chain Monte Carlo

We begin by giving a cursory overview of the key elements of Bayesian inference required for our purposes here.[8] In brief, the Bayesian approach stems from Bayes' theorem, in which the posterior distribution for an object of interest is derived as proportional to the combination of a prior distribution and the likelihood function

$$\underbrace{p(A|B)}_{\text{posterior}} \propto \underbrace{p(B|A)}_{\text{likelihood}} \times \underbrace{p(A)}_{\text{prior}}$$

Here, we will be interested in the posterior distribution of the parameters of our model and of the unobserved states, conditional on the chosen model specification and the observed time series data. While in most cases the form of the posterior cannot be derived analytically, simulation-based methods such as Markov chain Monte Carlo (MCMC) can be used to draw samples that approximate the posterior distribution nonetheless. While PyMC3, PyStan, and TensorFlow Probability emphasize Hamiltonian Monte Carlo (HMC) and no-U-turn sampling (NUTS) MCMC methods, we focus on the simpler random walk Metropolis-Hastings (MH) and Gibbs sampling (GS) methods. These are standard MCMC methods that have enjoyed great success in time series applications and which are simple to implement, given the state space framework already available in `statsmodels`. In addition, the ArviZ library is designed to work with MCMC output from any source, and we can easily adapt it to our use.

With either Metropolis-Hastings or Gibbs sampling, our procedure will produce a sequence of sample values (of parameters and / or the unobserved state vector) that approximate draws from the posterior distribution arbitrarily well, as the number of length

7. `Statsmodels` currently contains two implementations of simulation smoothers for the linear Gaussian state space model. The default is the "mean correction" simulation smoother of [DK02]. The precision-based simulation smoother of [CJ09] can alternatively be used by specifying `method='cfa'` when creating the simulation smoother object.

of the chain of samples becomes very large.

Random walk Metropolis-Hastings

In random walk Metropolis-Hastings (MH), we begin with an arbitrary point as the initial sample, and then iteratively construct new samples in the chain as follows. At each iteration, (a) construct a proposal by perturbing the previous sample by a Gaussian random variable, and then (b) accept the proposal with some probability. If a proposal is accepted, it becomes the next sample in the chain, while if it is rejected then the previous sample value is carried over. Here, we show how to implement Metropolis-Hastings estimation of the variance parameter in a simple model, which only requires the use of the log-likelihood computation introduced above.

```
import arviz as az
from scipy import stats

# Construct the model
model_rw = sm.tsa.UnobservedComponents(y, 'rwalk')

# Specify the prior distribution. With MH, this
# can be freely chosen by the user
prior = stats.uniform(0.0001, 100)

# Specify the Gaussian perturbation distribution
perturb = stats.norm(scale=0.1)

# Storage
niter = 100000
samples_rw = np.zeros(niter + 1)

# Initialization
samples_rw[0] = y.diff().var()
llf = model_rw.loglike(samples_rw[0])
prior_llf = prior.logpdf(samples_rw[0])

# Iterations
for i in range(1, niter + 1):
    # Compute the proposal value
    proposal = samples_rw[i - 1] + perturb.rvs()

    # Compute the acceptance probability
    proposal_llf = model_rw.loglike(proposal)
    proposal_prior_llf = prior.logpdf(proposal)
    accept_prob = np.exp(
        proposal_llf - llf
        + prior_llf - proposal_prior_llf)

    # Accept or reject the value
    if accept_prob > stats.uniform.rvs():
        samples_rw[i] = proposal
        llf = proposal_llf
        prior_llf = proposal_prior_llf
    else:
        samples_rw[i] = samples_rw[i - 1]

# Convert for use with ArviZ and plot posterior
samples_rw = az.convert_to_inference_data(
    samples_rw)
# Eliminate the first 10000 samples as burn-in;
# thin by factor of 10 to reduce autocorrelation
az.plot_posterior(samples_rw.posterior.sel(
    {'draw': np.s_[10000::10]}), kind='bin',
    point_estimate='median')
```

The approximate posterior distribution, constructed from the sample chain, is shown in Figure 2.

8. While a detailed description of these issues is out of the scope of this paper, there are many superb references on this topic. We refer the interested reader to [WH99], which provides a book-length treatment of Bayesian inference for state space models, and [KN99], which provides many examples and applications.

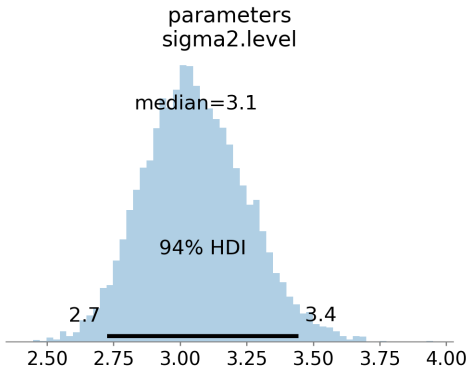


Fig. 2: *Approximate posterior distribution of variance parameter, random walk model, Metropolis-Hastings; U.S. Industrial Production.*

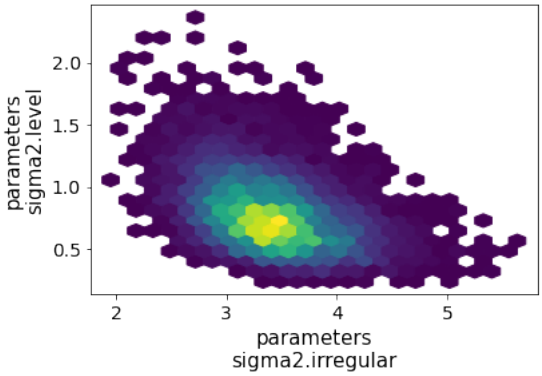


Fig. 3: *Approximate posterior joint distribution of variance parameters, local level model, Gibbs sampling; CPI inflation.*

Gibbs sampling

Gibbs sampling (GS) is a special case of Metropolis-Hastings (MH) that is applicable when it is possible to produce draws directly from the conditional distributions of every variable, even though it is still not possible to derive the general form of the joint posterior. While this approach can be superior to random walk MH when it is applicable, the ability to derive the conditional distributions typically requires the use of a "conjugate" prior – i.e., a prior from some specific family of distributions. For example, above we specified a uniform distribution as the prior when sampling via MH, but that is not possible with Gibbs sampling. Here, we show how to implement Gibbs sampling estimation of the variance parameter, now making use of an inverse Gamma prior, and the simulation smoother introduced above.

```
# Construct the model and simulation smoother
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')
sim_ll = model_ll.simulation_smoother()

# Specify the prior distributions. With GS, we must
# choose an inverse Gamma prior for each variance
priors = [stats.invgamma(0.01, scale=0.01)] * 2

# Storage
niter = 100000
samples_ll = np.zeros((niter + 1, 2))

# Initialization
samples_ll[0] = [y.diff().var(), 1e-5]

# Iterations
```

```
for i in range(1, niter + 1):
    # (a) Update the model parameters
    model_ll.update(samples_ll[i - 1])

    # (b) Draw from the conditional posterior of
    # the state vector
    sim_ll.simulate()
    sample_state = sim_ll.simulated_state.T

    # (c) Compute / draw from conditional posterior
    # of the parameters:
    # ...observation error variance
    resid = y - sample_state[:, 0]
    post_shape = len(resid) / 2 + 0.01
    post_scale = np.sum(resid**2) / 2 + 0.01
    samples_ll[i, 0] = stats.invgamma(
        post_shape, scale=post_scale).rvs()

    # ...level error variance
    resid = sample_state[1:] - sample_state[:-1]
    post_shape = len(resid) / 2 + 0.01
    post_scale = np.sum(resid**2) / 2 + 0.01
    samples_ll[i, 1] = stats.invgamma(
        post_shape, scale=post_scale).rvs()

# Convert for use with ArviZ and plot posterior
samples_ll = az.convert_to_inference_data(
    {'parameters': samples_ll[None, ...]},
    coords={'parameter': model_ll.param_names},
    dims={'parameters': ['parameter']})
az.plot_pair(samples_ll.posterior.sel(
    {'draw': np.s_[10000::10]}), kind='hexbin');
```

The approximate posterior distribution, constructed from the sample chain, is shown in Figure 3.

Illustrative examples

For clarity and brevity, the examples in the previous section gave results for simple cases. However, these basic methods carry through to each of the models introduced earlier, including in cases with multivariate data and hundreds of parameters. Moreover, the Metropolis-Hastings approach can be combined with the Gibbs sampling approach, so that if the end user wishes to use Gibbs sampling for some parameters, they are not restricted to choose only conjugate priors for all parameters.

In addition to sampling the posterior distributions of the parameters, this method allows sampling other objects of interest, including forecasts of observed variables, impulse response functions, and the unobserved state vector. This last possibility is especially useful in cases such as the structural time series model, in which the unobserved states correspond to interpretable elements such as the trend and seasonal components. We provide several illustrative examples of the various types of analysis that are possible.

Forecasting and Time Series Decomposition

In our first example, we apply the Gibbs sampling approach to a structural time series model in order to forecast U.S. Industrial Production and to produce a decomposition of the series into level, trend, and seasonal components. The model is

$$
\begin{aligned}
y_t &= \mu_t + \gamma_t + \varepsilon_t && \text{observation equation} \\
\mu_t &= \beta_t + \mu_{t-1} + \zeta_t && \text{level} \\
\beta_t &= \beta_{t-1} + \xi_t && \text{trend} \\
\gamma_t &= \gamma_{t-s} + \eta_t && \text{seasonal}
\end{aligned}
$$

Here, we set the seasonal periodicity to *s=12*, since Industrial Production is a monthly variable. We can construct this model in Statsmodels as[9]

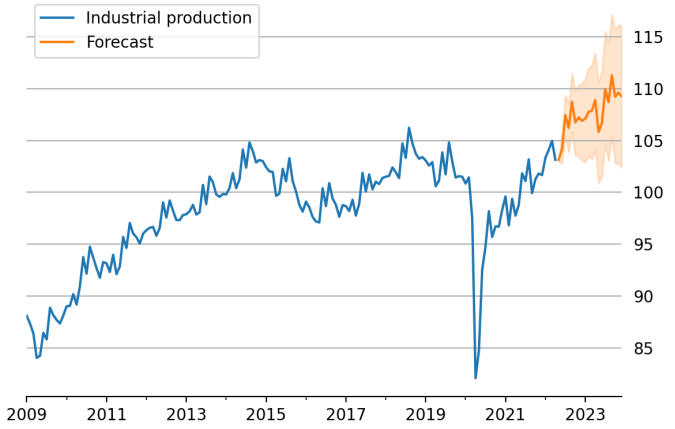


Fig. 4: *Data and forecast with 80% credible interval; U.S. Industrial Production.*

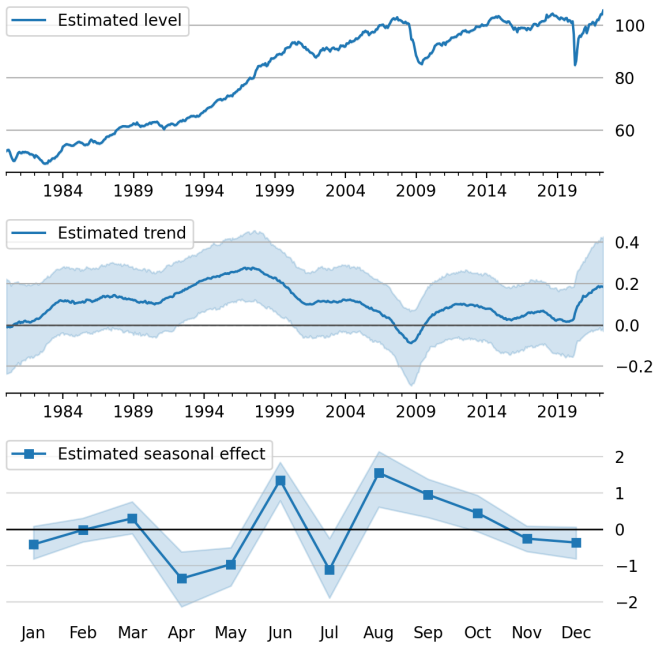


Fig. 5: *Estimated level, trend, and seasonal components, with 80% credible interval; U.S. Industrial Production.*

```
model = sm.tsa.UnobservedComponents(
    y, 'lltrend', seasonal=12)
```

To produce the time-series decomposition into level, trend, and seasonal components, we will use samples from the posterior of the state vector $(\mu_t, \beta_t, \gamma_t)$ for each time period t. These are immediately available when using the Gibbs sampling approach; in the earlier example, the draw at each iteration was assigned to the variable *sample_state*. To produce forecasts, we need to draw from the posterior predictive distribution for horizons $h = 1, 2, \ldots H$. This can be easily accomplished by using the *simulate* method introduced earlier. To be concrete, we can accomplish these tasks by modifying section (b) of our Gibbs sampler iterations as follows:

9. This model is often referred to as a "local linear trend" model (with additionally a seasonal component); *lltrend* is an abbreviation of this name.

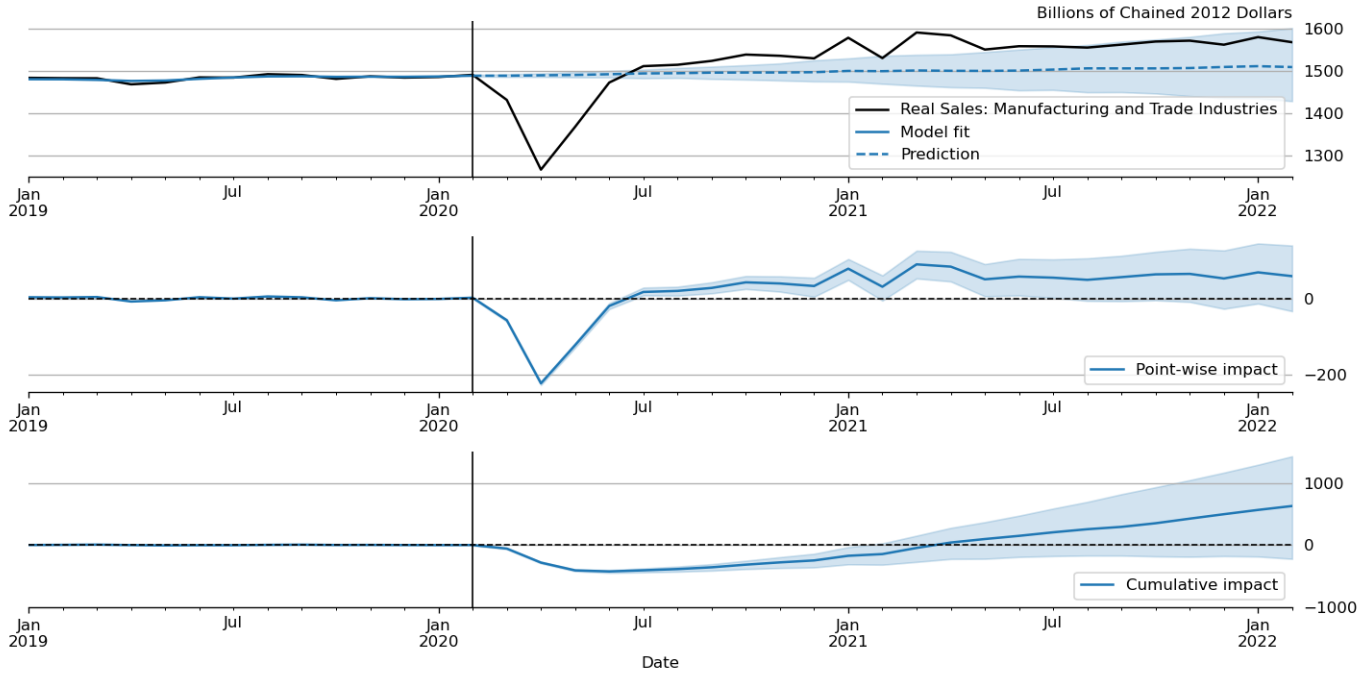


Fig. 6: *"Causal impact" of COVID-19 on U.S. Sales in Manufacturing and Trade Industries.*

```
# (b') Draw from the conditional posterior of
# the state vector
model.update(params[i - 1])
sim.simulate()
# save the draw for use later in time series
# decomposition
states[i] = sim.simulated_state.T

# Draw from the posterior predictive distribution
# using the `simulate` method
n_fcast = 48
fcast[i] = model.simulate(
   params[i - 1], n_fcast,
   initial_state=states[i, -1]).to_frame()
```

These forecasts and the decomposition into level, trend, and seasonal components are summarized in Figures 4 and 5, which show the median values along with 80% credible intervals. Notably, the intervals shown incorporate for both the uncertainty arising from the stochastic terms in the model as well as the need to estimate the models' parameters.[10]

Casual impacts

A closely related procedure described in [BGK+15] uses a Bayesian structural time series model to estimate the "causal impact" of some event on some observed variable. This approach stops estimation of the model just before the date of an event and produces a forecast by drawing from the posterior predictive density, using the procedure described just above. It then uses the difference between the actual path of the data and the forecast to estimate impact of the event.

An example of this approach is shown in Figure 6, in which we use this method to illustrate the effect of the COVID-19 pandemic on U.S. Sales in Manufacturing and Trade Industries.[11]

Extensions

There are many extensions to the time series models presented here that are made possible when using Bayesian inference. First, it is easy to create custom state space models within the `statsmodels` framework. As one example, the `statsmodels` documentation describes how to create a model that extends the typical VAR described above with time-varying parameters.[12] These custom state space models automatically inherit all the functionality described above, so that Bayesian inference can be conducted in exactly the same way.

Second, because the general state space model available in `statsmodels` and introduced above allows for time-varying system matrices, it is possible using Gibbs sampling methods to introduce support for automatic outlier handling, stochastic volatility, and regime switching models, even though these are largely infeasible in `statsmodels` when using frequentist methods such as maximum likelihood estimation.[13]

Conclusion

This paper introduces the suite of time series models available in `statsmodels` and shows how Bayesian inference using Markov chain Monte Carlo methods can be applied to estimate their parameters and produce analyses of interest, including time series decompositions and forecasts.

10. The popular Prophet library, [TL17], similarly uses an additive model combined with Bayesian sampling methods to produce forecasts and decompositions, although its underlying model is a GAM rather than a state space model.

11. In this example, we used a local linear trend model with no seasonal component.

12. For details, see https://www.statsmodels.org/devel/examples/notebooks/generated/statespace_tvpvar_mcmc_cfa.html.

13. See, for example, [SW16] for an application of these techniques that handles outliers, [KSC98] for stochastic volatility, and [KN98] for an application to dynamic factor models with regime switching.

REFERENCES

[BGK+15] Kay H. Brodersen, Fabian Gallusser, Jim Koehler, Nicolas Remy, and Steven L. Scott. Inferring causal impact using Bayesian structural time-series models. *Annals of Applied Statistics*, 9:247–274, 2015. doi:10.1214/14-aoas788.

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan : A Probabilistic Programming Language. *Journal of Statistical Software*, 76(1), January 2017. Institution: Columbia Univ., New York, NY (United States); Harvard Univ., Cambridge, MA (United States). URL: https://www.osti.gov/pages/biblio/1430202-stan-probabilistic-programming-language, doi:10.18637/jss.v076.i01.

[CJ09] Joshua C.C. Chan and Ivan Jeliazkov. Efficient simulation and integrated likelihood estimation in state space models. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(1-2):101–120, January 2009. Publisher: Inderscience Publishers. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJMMNO.2009.03009.

[DK02] J. Durbin and S. J. Koopman. A simple and efficient simulation smoother for state space time series analysis. *Biometrika*, 89(3):603–616, August 2002. URL: http://biomet.oxfordjournals.org/content/89/3/603, doi:10.1093/biomet/89.3.603.

[DK12] James Durbin and Siem Jan Koopman. *Time Series Analysis by State Space Methods: Second Edition*. Oxford University Press, May 2012.

[DLT+17] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A. Saurous. TensorFlow Distributions. Technical Report arXiv:1711.10604, arXiv, November 2017. arXiv:1711.10604 [cs, stat] type: article. URL: http://arxiv.org/abs/1711.10604, doi:10.48550/arXiv.1711.10604.

[Ful15] Chad Fulton. Estimating time series models by state space methods in python: Statsmodels. 2015.

[HA18] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[Har90] Andrew C. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1990.

[HKOS08] Rob Hyndman, Anne B. Koehler, J. Keith Ord, and Ralph D. Snyder. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Science & Business Media, June 2008. Google-Books-ID: GSyzox8Lu9YC.

[KCHM19] Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *Journal of Open Source Software*, 4(33):1143, 2019. Publisher: The Open Journal. URL: https://doi.org/10.21105/joss.01143, doi:10.21105/joss.01143.

[KN98] Chang-Jin Kim and Charles R. Nelson. Business Cycle Turning Points, A New Coincident Index, and Tests of Duration Dependence Based on a Dynamic Factor Model With Regime Switching. *The Review of Economics and Statistics*, 80(2):188–201, May 1998. Publisher: MIT Press. URL: https://doi.org/10.1162/003465398557447, doi:10.1162/003465398557447.

[KN99] Chang-Jin Kim and Charles R. Nelson. State-Space Models with Regime Switching: Classical and Gibbs-Sampling Approaches with Applications. MIT Press Books, The MIT Press, 1999. URL: http://ideas.repec.org/b/mtp/titles/0262112388.html.

[KSC98] Sangjoon Kim, Neil Shephard, and Siddhartha Chib. Stochastic Volatility: Likelihood Inference and Comparison with ARCH Models. *The Review of Economic Studies*, 65(3):361–393, July 1998. 01855. URL: http://restud.oxfordjournals.org/content/65/3/361, doi:10.1111/1467-937X.00050.

[MPS11] Wes McKinney, Josef Perktold, and Skipper Seabold. Time Series Analysis in Python with statsmodels. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 10th Python in Science Conference*, pages 107 – 113, 2011. doi:10.25080/Majora-ebaa42b7-012.

[SP10] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and Statistical Modeling with Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 92 – 96, 2010. doi:10.25080/Majora-92bf1922-011.

[SW16] James H. Stock and Mark W. Watson. Core Inflation and Trend Inflation. *Review of Economics and Statistics*, 98(4):770–784, March 2016. 00000. URL: http://dx.doi.org/10.1162/REST_a_00608, doi:10.1162/REST_a_00608.

[SWF16] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, April 2016. Publisher: PeerJ Inc. URL: https://peerj.com/articles/cs-55, doi:10.7717/peerj-cs.55.

[TL17] Sean J. Taylor and Benjamin Letham. Forecasting at scale. Technical Report e3190v2, PeerJ Inc., September 2017. ISSN: 2167-9843. URL: https://peerj.com/preprints/3190, doi:10.7287/peerj.preprints.3190v2.

[WH99] Mike West and Jeff Harrison. *Bayesian Forecasting and Dynamic Models*. Springer, New York, 2nd edition edition, March 1999. 00000.

Python vs. the pandemic: a case study in high-stakes software development

Cliff C. Kerr[‡§*], Robyn M. Stuart[¶‖], Dina Mistry[**], Romesh G. Abeysuriya[‖], Jamie A. Cohen[‡], Lauren George[††], Michał Jastrzebski[‡‡], Michael Famulare[‡], Edward Wenger[‡], Daniel J. Klein[‡]

Abstract—When it became clear in early 2020 that COVID-19 was going to be a major public health threat, politicians and public health officials turned to academic disease modelers like us for urgent guidance. Academic software development is typically a slow and haphazard process, and we realized that business-as-usual would not suffice for dealing with this crisis. Here we describe the case study of how we built Covasim (covasim.org), an agent-based model of COVID-19 epidemiology and public health interventions, by using standard Python libraries like NumPy and Numba, along with less common ones like Sciris (sciris.org). Covasim was created in a few weeks, an order of magnitude faster than the typical model development process, and achieves performance comparable to C++ despite being written in pure Python. It has become one of the most widely adopted COVID models, and is used by researchers and policymakers in dozens of countries. Covasim's rapid development was enabled not only by leveraging the Python scientific computing ecosystem, but also by adopting coding practices and workflows that lowered the barriers to entry for scientific contributors without sacrificing either performance or rigor.

Index Terms—COVID-19, SARS-CoV-2, Epidemiology, Mathematical modeling, NumPy, Numba, Sciris

Background

For decades, scientists have been concerned about the possibility of another global pandemic on the scale of the 1918 flu [Gar05]. Despite a number of "close calls" – including SARS in 2002 [AFG+04]; Ebola in 2014-2016 [Tea14]; and flu outbreaks including 1957, 1968, and H1N1 in 2009 [SHK16], some of which led to 1 million or more deaths – the last time we experienced the emergence of a planetary-scale new pathogen was when HIV spread globally in the 1980s [CHL+08].

In 2015, Bill Gates gave a TED talk stating that the world was not ready to deal with another pandemic [Hof20]. While the Bill & Melinda Gates Foundation (BMGF) has not historically focused on pandemic preparedness, its expertise in disease surveillance, modeling, and drug discovery made it well placed to contribute to a global pandemic response plan. Founded in 2008, the Institute for Disease Modeling (IDM) has provided analytical support for BMGF (which it has been a part of since 2020) and other global health partners, with a focus on eradicating malaria and polio. Since its creation, IDM has built up a portfolio of computational tools to understand, analyze, and predict the dynamics of different diseases.

When "coronavirus disease 2019" (COVID-19) and the virus that causes it (SARS-CoV-2) were first identified in late 2019, our team began summarizing what was known about the virus [Fam19]. By early February 2020, even though it was more than a month before the World Health Organization (WHO) declared a pandemic [Med20], it had become clear that COVID-19 would become a major public health threat. The outbreak on the *Diamond Princess* cruise ship [RSWS20] was the impetus for us to start modeling COVID in detail. Specifically, we needed a tool to (a) incorporate new data as soon as it became available, (b) explore policy scenarios, and (c) predict likely future epidemic trajectories.

The first step was to identify which software tool would form the best starting point for our new COVID model. Infectious disease models come in two major types: *agent-based models* track the behavior of individual "people" (agents) in the simulation, with each agent's behavior represented by a random (probabilistic) process. *Compartmental models* track populations of people over time, typically using deterministic difference equations. The richest modeling framework used by IDM at the time was EMOD, which is a multi-disease agent-based model written in C++ and based on JSON configuration files [BGB+18]. We also considered Atomica, a multi-disease compartmental model written in Python and based on Excel input files [KAK+19]. However, both of these options posed significant challenges: as a compartmental model, Atomica would have been unable to capture the individual-level detail necessary for modeling the *Diamond Princess* outbreak (such as passenger-crew interactions); EMOD had sufficient flexibility, but developing new disease modules had historically required months rather than days.

As a result, we instead started developing Covasim ("**COV**ID-19 **A**gent-based **Sim**ulator") [KSM+21] from a nascent agent-based model written in Python, LEMOD-FP ("Light-EMOD for Family Planning"). LEMOD-FP was used to model reproductive health choices of women in Senegal; this model had in turn been based on an even simpler agent-based model of measles vaccination programs in Nigeria ("Value-of-Information Simulator" or VoISim). We subsequently applied the lessons we learned

* *Corresponding author: cliff@covasim.org*
‡ *Institute for Disease Modeling, Bill & Melinda Gates Foundation, Seattle, USA*
§ *School of Physics, University of Sydney, Sydney, Australia*
¶ *Department of Mathematical Sciences, University of Copenhagen, Copenhagen, Denmark*
‖ *Burnet Institute, Melbourne, Australia*
** *Twitter, Seattle, USA*
†† *Microsoft, Seattle, USA*
‡‡ *GitHub, San Francisco, USA*

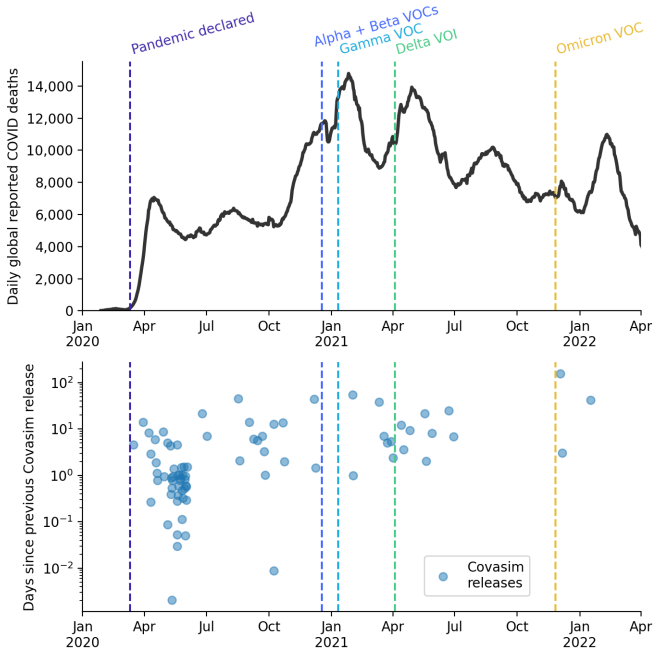


Fig. 1: *Daily reported global COVID-19-related deaths (top; smoothed with a one-week rolling window), relative to the timing of known variants of concern (VOCs) and variants of interest (VOIs), as well as Covasim releases (bottom).*

from developing Covasim to turn LEMOD-FP into a new family planning model, "FPsim", which will be launched later this year [OVCC+22].

Parallel to the development of Covasim, other research teams at IDM developed their own COVID models, including one based on the EMOD framework [SWC+22], and one based on an earlier influenza model [COSF20]. However, while both of these models saw use in academic contexts [KCP+20], neither were able to incorporate new features quickly enough, or were easy enough to use, for widespread external adoption in a policy context.

Covasim, by contrast, had immediate real-world impact. The first version was released on 10 March 2020, and on 12 March 2020, its output was presented by Washington State Governor Jay Inslee during a press conference as justification for school closures and social distancing measures [KMS+21].

Since the early days of the pandemic, Covasim releases have coincided with major events in the pandemic, especially the identification of new variants of concern (Fig. 1). Covasim was quickly adopted globally, including applications in the UK regarding school closures [PGKS+20], Australia regarding outbreak control [SAK+21], and Vietnam regarding lockdown measures [PSN+21].

To date, Covasim has been downloaded from PyPI over 100,000 times [PeP22], has been used in dozens of academic studies [KMS+21], and informed decision-making on every continent (Fig. 2), making it one of the most widely used COVID models [KSM+21]. We believe key elements of its success include (a) the simplicity of its architecture; (b) its high performance, enabled by the use of NumPy arrays and Numba decorators; and (c) our emphasis on prioritizing usability, including flexible type handling and careful choices of default settings. In the remainder of this paper, we outline these principles in more detail, in the hope that these will provide a useful roadmap for other groups wanting to quickly develop high-performance, easy-to-use scientific computing libraries.

Software architecture and implementation

Covasim conceptual design and usage

Covasim is a standard susceptible-exposed-infectious-recovered (SEIR) model (Fig. 3). As noted above, it is an agent-based model, meaning that individual people and their interactions with one another are simulated explicitly (rather than implicitly, as in a compartmental model).

The fundamental calculation that Covasim performs is to determine the probability that a given person, on a given time step, will change from one state to another, such as from susceptible to exposed (i.e., that person was infected), from undiagnosed to diagnosed, or from critically ill to dead. Covasim is fully open-source and available on GitHub (http://covasim.org) and PyPI (`pip install covasim`), and comes with comprehensive documentation, including tutorials (http://docs.covasim.org).

The first principle of Covasim's design philosophy is that "Common tasks should be simple" – for example, defining parameters, running a simulation, and plotting results. The following example illustrates this principle; it creates a simulation with a custom parameter value, runs it, and plots the results:

```
import covasim as cv
cv.Sim(pop_size=100e3).run().plot()
```

The second principle of Covasim's design philosophy is "Uncommon tasks can't always be simple, but they still should be possible." Examples include writing a custom goodness-of-fit function or defining a new population structure. To some extent, the second principle is at odds with the first, since the more flexibility an interface has, typically the more complex it is as well.

To illustrate the tension between these two principles, the following code shows how to run two simulations to determine the impact of a custom intervention aimed at protecting the elderly in Japan, with results shown in Fig. 4:

```
import covasim as cv

# Define a custom intervention
def elderly(sim, old=70):
    if sim.t == sim.day('2020-04-01'):
        elderly = sim.people.age > old
        sim.people.rel_sus[elderly] = 0.0

# Set custom parameters
pars = dict(
    pop_type = 'hybrid', # More realistic population
    location = 'japan', # Japan's population pyramid
    pop_size = 50e3, # Have 50,000 people total
    pop_infected = 100, # 100 infected people
    n_days = 90, # Run for 90 days
)

# Run multiple sims in parallel and plot key results
label = 'Protect the elderly'
s1 = cv.Sim(pars, label='Default')
s2 = cv.Sim(pars, interventions=elderly, label=label)
msim = cv.parallel(s1, s2)
msim.plot(['cum_deaths', 'cum_infections'])
```

Similar design philosophies have been articulated by previously, such as for Grails [AJ09] among others[1].

1. Other similar philosophical statements include "The manifesto of Matplotlib is: simple and common tasks should be simple to perform; provide options for more complex tasks" (Data Processing Using Python) and "Simple, common tasks should be simple to perform; Options should be provided to enable more complex tasks" (Instrumental).

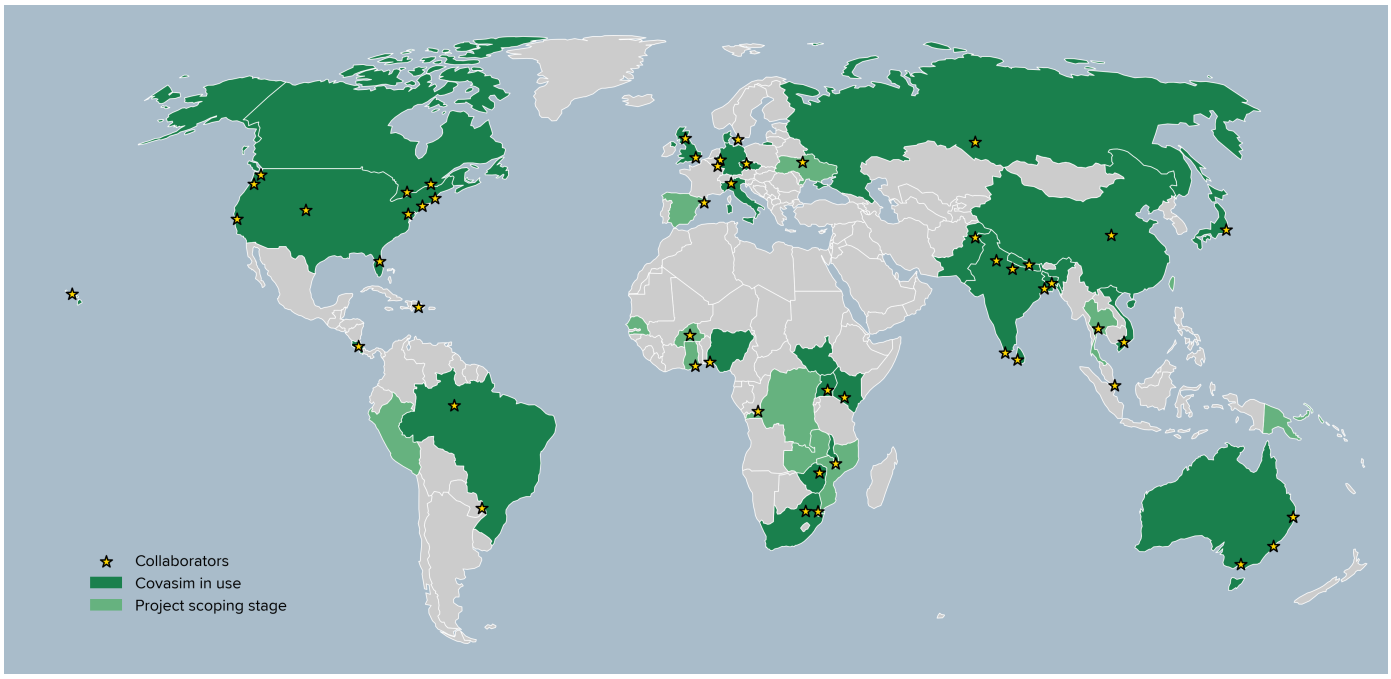


Fig. 2: *Locations where Covasim has been used to help produce a paper, report, or policy recommendation.*

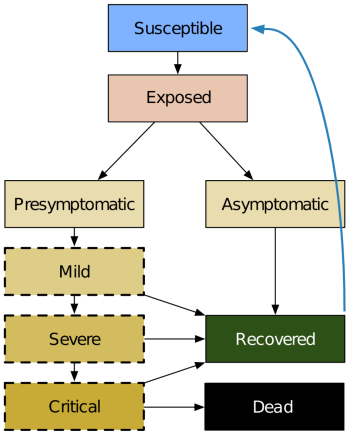


Fig. 3: *Basic Covasim disease model. The blue arrow shows the process of reinfection.*

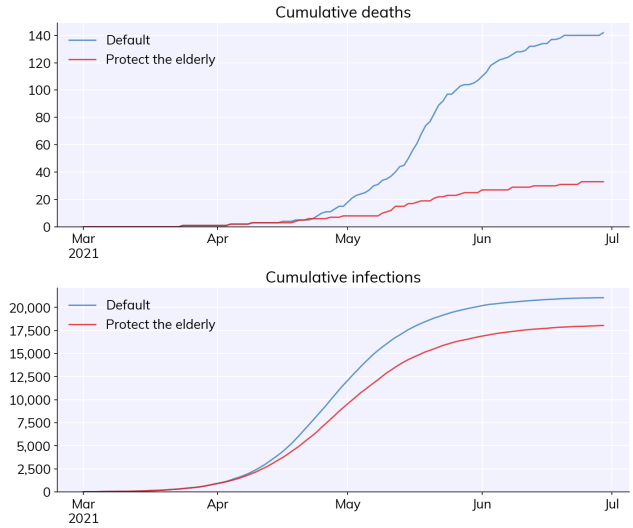


Fig. 4: *Illustrative result of a simulation in Covasim focused on exploring an intervention for protecting the elderly.*

Simplifications using Sciris

A key component of Covasim's architecture is heavy reliance on Sciris (http://sciris.org) [KAH+ng], a library of functions for scientific computing that provide additional flexibility and ease-of-use on top of NumPy, SciPy, and Matplotlib, including parallel computing, array operations, and high-performance container datatypes.

As shown in Fig. 5, Sciris significantly reduces the number of lines of code required to perform common scientific tasks, allowing the user to focus on the code's scientific logic rather than the low-level implementation. Key Covasim features that rely on Sciris include: ensuring consistent dictionary, list, and array types (e.g., allowing the user to provide inputs as either lists or arrays); referencing ordered dictionary elements by index; handling and interconverting dates (e.g., allowing the user to provide either a date string or a `datetime` object); saving and loading files; and running simulations in parallel.

Array-based architecture

In a typical agent-based simulation, the outermost loop is over time, while the inner loops iterate over different agents and agent states. For a simulation like Covasim, with roughly 700 (daily) timesteps to represent the first two years of the pandemic, tens or hundreds of thousands of agents, and several dozen states, this requires on the order of one billion update steps.

However, we can take advantage of the fact that each state (such as agent age or their infection status) has the same data type, and thus we can avoid an explicit loop over agents by instead representing agents as entries in NumPy vectors, and performing operations on these vectors. These two architectures are shown in

```
# Set parameters and define random wave generator
import numpy as np
xmin = 0
xmax = 10
npts = 50
noisevals = np.linspace(0, 1, 11)

def randwave(std):
    np.random.seed() # Ensure differences between runs
    a = np.cos(np.linspace(xmin, xmax, npts))
    b = np.random.randn(npts)
    return a + b*std

# Other imports
- import time
- import multiprocessing as mp
- import pickle
- import gzip
- import matplotlib.pyplot as plt
- from mpl_toolkits.mplot3d import Axes3D # Unused but must be imported

# Start timing
- start = time.time()

# Create object in parallel
- multipool = mp.Pool(processes=mp.cpu_count())
- output = multipool.map(randwave, noisevals)
- multipool.close()
- multipool.join()

# Save to files
filenames = []
for n,noiseval in enumerate(noisevals):
    filename = f'noise{n}.obj'
-     with gzip.GzipFile(filename, 'wb') as fileobj:
-         fileobj.write(pickle.dumps(output[n]))
    filenames.append(filename)

# Create dict from files
- data = {}
- for filename in filenames:
-     with gzip.GzipFile(filename) as fileobj:
-         filestring = fileobj.read()
-         data[filename] = pickle.loads(filestring)

# Create 3D plot
- data_array = np.array([data[filename] for filename in filenames])
- fig = plt.figure()
- ax = plt.axes(projection='3d')
- ax.view_init(elev=45, azim=30)
- ny,nx = np.array(data_array).shape
- x = np.arange(nx)
- y = np.arange(ny)
- X, Y = np.meshgrid(x, y)
- surf = ax.plot_surface(X, Y, data_array, cmap='viridis')
- fig.colorbar(surf)

# Print elapsed time
- elapsed = time.time() - start
- print(f'Elapsed time: {elapsed:0.1f} s')
```

```
# Set parameters and define random wave generator
import numpy as np
xmin = 0
xmax = 10
npts = 50
noisevals = np.linspace(0, 1, 11)

def randwave(std):
    np.random.seed() # Ensure differences between runs
    a = np.cos(np.linspace(xmin, xmax, npts))
    b = np.random.randn(npts)
    return a + b*std

# Other imports
+ import sciris as sc

# Start timing
+ sc.tic()

# Create object in parallel
+ output = sc.parallelize(randwave, noisevals)

# Save to files
filenames = []
for n,noiseval in enumerate(noisevals):
    filename = f'noise{n}.obj'
+     sc.save(filename, output[n])
    filenames.append(filename)

# Create dict from files
+ data = sc.odict({filename:sc.load(filename) for filename in filenames})

# Create 3D plot
+ sc.surf3d(data[:])

# Print elapsed time
+ sc.toc()
```

Fig. 5: *Comparison of functionally identical code implemented without Sciris (left) and with (right). In this example, tasks that together take 30 lines of code without Sciris can be accomplished in 7 lines with it.*

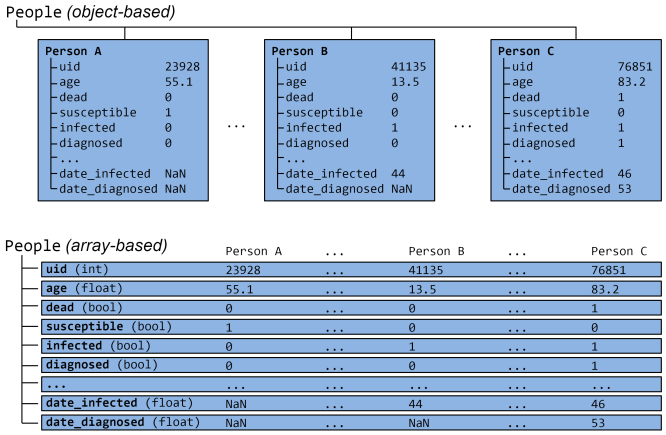


Fig. 6: *The standard object-oriented approach for implementing agent-based models (top), compared to the array-based approach used in Covasim (bottom).*

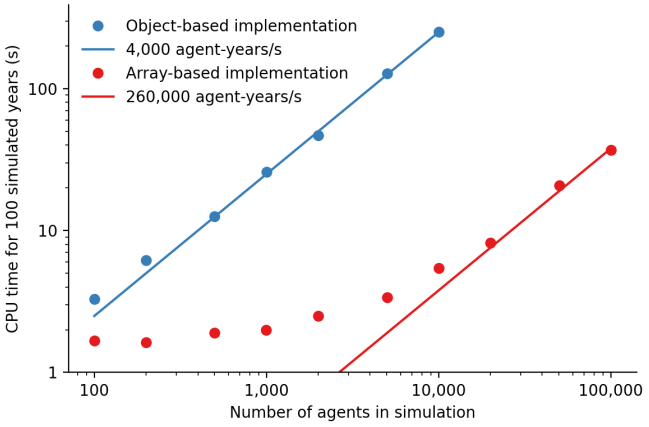


Fig. 7: *Performance comparison for FPsim from an explicit loop-based approach compared to an array-based approach, showing a factor of ~70 speed improvement for large population sizes.*

Fig. 6. Compared to the explicitly object-oriented implementation of an agent-based model, the array-based version is 1-2 orders of magnitude faster for population sizes larger than 10,000 agents. The relative performance of these two approaches is shown in Fig. 7 for FPsim (which, like Covasim, was initially implemented using an object-oriented approach before being converted to an array-based approach). To illustrate the difference between object-based and array-based implementations, the following example shows how aging and death would be implemented in each:

```
# Object-based agent simulation

class Person:

    def age_person(self):
        self.age += 1
        return

    def check_died(self):
        rand = np.random.random()
        if rand < self.death_prob:
            self.alive = False
        return

class Sim:

    def run(self):
        for t in self.time_vec:
            for person in self.people:
                if person.alive:
                    person.age_person()
                    person.check_died()

# Array-based agent simulation

class People:

    def age_people(self, inds):
        self.age[inds] += 1
        return

    def check_died(self, inds):
        rands = np.random.rand(len(inds))
        died = rands < self.death_probs[inds]:
        self.alive[inds[died]] = False
        return

class Sim:

    def run(self):
        for t in self.time_vec:
            alive = sc.findinds(self.people.alive)
            self.people.age_people(inds=alive)
            self.people.check_died(inds=alive)
```

Numba optimization

Numba is a compiler that translates subsets of Python and NumPy into machine code [LPS15]. Each low-level numerical function was tested with and without Numba decoration; in some cases speed improvements were negligible, while in other cases they were considerable. For example, the following function is roughly 10 times faster with the Numba decorator than without:

```
import numpy as np
import numba as nb

@nb.njit((nb.int32, nb.int32), cache=True)
def choose_r(max_n, n):
    return np.random.choice(max_n, n, replace=True)
```

Since Covasim is stochastic, calculations rarely need to be exact; as a result, most numerical operations are performed as 32-bit operations.

Together, these speed optimizations allow Covasim to run at roughly 5-10 million simulated person-days per second of CPU time – a speed comparable to agent-based models implemented purely in C or C++ [HPN+21]. Practically, this means that most users can run Covasim analyses on their laptops without needing to use cloud-based or HPC computing resources.

Lessons for scientific software development

Accessible coding and design

Since Covasim was designed to be used by scientists and health officials, not developers, we made a number of design decisions that preferenced accessibility to our audience over other principles of good software design.

First, Covasim is designed to have as flexible of user inputs as possible. For example, a date can be specified as an integer number of days from the start of the simulation, as a string (e.g. `'2020-04-04'`), or as a `datetime` object. Similarly, numeric inputs that can have either one or multiple values (such as the change in transmission rate following one or multiple lockdowns) can be provided as a scalar, list, or NumPy array. As long as the input is unambiguous, we prioritized ease-of-use and simplicity of the interface over rigorous type checking. Since Covasim is a

top-level library (i.e., it does not perform low-level functions as part of other libraries), this prioritization has been welcomed by its users.

Second, "advanced" Python programming paradigms – such as method and function decorators, lambda functions, multiple inheritance, and "dunder" methods – have been avoided where possible, even when they would otherwise be good coding practice. This is because a relatively large fraction of Covasim users, including those with relatively limited Python backgrounds, need to inspect and modify the source code. A Covasim user coming from an R programming background, for example, may not have encountered the NumPy function `intersect1d()` before, but they can quickly look it up and understand it as being equivalent to R's `intersect()` function. In contrast, an R user who has not encountered method decorators before is unlikely to be able to look them up and understand their meaning (indeed, they may not even know what terms to search for). While Covasim indeed does use each of the "advanced" methods listed above (e.g., the Numba decorators described above), they have been kept to a minimum and sequestered in particular files the user is less likely to interact with.

Third, testing for Covasim presented a major challenge. Given that Covasim was being used to make decisions that affected tens of millions of people, even the smallest errors could have potentially catastrophic consequences. Furthermore, errors could arise not only in the software logic, but also in an incorrectly entered parameter value or a misinterpreted scientific study. Compounding these challenges, features often had to be developed and used on a timescale of hours or days to be of use to policymakers, a speed which was incompatible with traditional software testing approaches. In addition, the rapidly evolving codebase made it difficult to write even simple regression tests. Our solution was to use a hierarchical testing approach: low-level functions were tested through a standard software unit test approach, while new features and higher-level outputs were tested extensively by infectious disease modelers who varied inputs corresponding to realistic scenarios, and checked the outputs (predominantly in the form of graphs) against their intuition. We found that these high-level "sanity checks" were far more effective in catching bugs than formal software tests, and as a result shifted the emphasis of our test suite to prioritize the former. Public releases of Covasim have held up well to extensive scrutiny, both by our external collaborators and by "COVID skeptics" who were highly critical of other COVID models [Den20].

Finally, since much of our intended audience has little to no Python experience, we provided as many alternative ways of accessing Covasim as possible. For R users, we provide examples of how to run Covasim using the `reticulate` package [AUTE17], which allows Python to be called from within R. For specific applications, such as our test-trace-quarantine work (http://ttq-app.covasim.org), we developed bespoke webapps via Jupyter notebooks [GP21] and Voilà [Qua19]. To help non-experts gain intuition about COVID epidemic dynamics, we also developed a generic JavaScript-based webapp interface for Covasim (http://app.covasim.org), but it does not have sufficient flexibility to answer real-world policy questions.

Workflow and team management

Covasim was developed by a team of roughly 75 people with widely disparate backgrounds: from those with 20+ years of enterprise-level software development experience and no public health background, through to public health experts with virtually no prior experience in Python. Roughly 45% of Covasim contributors had significant Python expertise, while 60% had public health experience; only about half a dozen contributors (<10%) had significant experience in both areas.

These half-dozen contributors formed a core group (including the authors of this paper) that oversaw overall Covasim development. Using GitHub for both software and project management, we created issues and assigned them to other contributors based on urgency and skillset match. All pull requests were reviewed by at least one person from this group, and often two, prior to merge. While the danger of accepting changes from contributors with limited Python experience is self-evident, considerable risks were also posed by contributors who lacked epidemiological insight. For example, some of the proposed tests were written based on assumptions that were true for a given time and place, but which were not valid for other geographical contexts.

One surprising outcome was that even though Covasim is largely a software project, after the initial phase of development (i.e., the first 4-8 weeks), we found that relatively few tasks could be assigned to the developers as opposed to the epidemiologists and infectious disease modelers on the project. We believe there are several reasons for this. First, epidemiologists tended to be much more aware of knowledge they were missing (e.g., what a particular NumPy function did), and were more readily able to fill that gap (e.g., look it up in the documentation or on Stack Overflow). By contrast, developers without expertise in epidemiology were less able to identify gaps in their knowledge and address them (e.g., by finding a study on Google Scholar). As a consequence, many of the epidemiologists' software skills improved markedly over the first few months, while the developers' epidemiology knowledge increased more slowly. Second, and more importantly, we found that once transparent and performant coding practices had been implemented, epidemiologists were able to successfully adapt them to new contexts even without complete understanding of the code. Thus, for developing a scientific software tool, we propose that a successful staffing plan would consist of a roughly equal ratio of developers and domain experts during the early development phase, followed by a rapid (on a timescale of weeks) ramp-down of developers and ramp-up of domain experts.

Acknowledging that Covasim's potential user base includes many people who have limited coding skills, we developed a three-tiered support model to maximize Covasim's real-world policy impact (Fig. 8). For "mode 1" engagements, we perform the analyses using Covasim ourselves. While this mode typically ensures high quality and efficiency, it is highly resource-constrained and thus used only for our highest-profile engagements, such as with the Vietnam Ministry of Health [PSN+21] and Washington State Department of Health [KMS+21]. For "mode 2" engagements, we offer our partners training on how to use Covasim, and let them lead analyses with our feedback. This is our preferred mode of engagement, since it balances efficiency and sustainability, and has been used for contexts including the United Kingdom [PGKS+20] and Australia [SLSS+22]. Finally, "mode 3" partnerships, in which Covasim is downloaded and used without our direct input, are of course the default approach in the open-source software ecosystem, including for Python. While this mode is by far the most scalable, in practice, relatively few health departments or ministries of health have the time and internal technical capacity to use this mode; instead, most of the mode 3 uptake of Covasim has

been by academic groups [LG+21]. Thus, we provide mode 1 and mode 2 partnerships to make Covasim's impact more immediate and direct than would be possible via mode 3 alone.

Future directions

While the need for COVID modeling is hopefully starting to decrease, we and our collaborators are continuing development of Covasim by updating parameters with the latest scientific evidence, implementing new immune dynamics [CSN+21], and providing other usability and bug-fix updates. We also continue to provide support and training workshops (including in-person workshops, which were not possible earlier in the pandemic).

We are using what we learned during the development of Covasim to build a broader suite of Python-based disease modeling tools (tentatively named "*-sim" or "Starsim"). The suite of Starsim tools under development includes models for family planning [OVCC+22], polio, respiratory syncytial virus (RSV), and human papillomavirus (HPV). To date, each tool in this suite uses an independent codebase, and is related to Covasim only through the shared design principles described above, and by having used the Covasim codebase as the starting point for development.

A major open question is whether the disease dynamics implemented in Covasim and these related models have sufficient overlap to be refactored into a single disease-agnostic modeling library, which the disease-specific modeling libraries would then import. This "core and specialization" approach was adopted by EMOD and Atomica, and while both frameworks continue to be used, no multi-disease modeling library has yet seen widespread adoption within the disease modeling community. The alternative approach, currently used by the Starsim suite, is for each disease model to be a self-contained library. A shared library would reduce code duplication, and allow new features and bug fixes to be immediately rolled out to multiple models simultaneously. However, it would also increase interdependencies that would have the effect of increasing code complexity, increasing the risk of introducing subtle bugs. Which of these two options is preferable likely depends on the speed with which new disease models need to be implemented. We hope that for the foreseeable future, none will need to be implemented as quickly as Covasim.

Acknowledgements

We thank additional contributors to Covasim, including Katherine Rosenfeld, Gregory R. Hart, Rafael C. Núñez, Prashanth Selvaraj, Brittany Hagedorn, Amanda S. Izzo, Greer Fowler, Anna Palmer, Dominic Delport, Nick Scott, Sherrie L. Kelly, Caroline S. Bennette, Bradley G. Wagner, Stewart T. Chang, Assaf P. Oron, Paula Sanz-Leon, and Jasmina Panovska-Griffiths. We also wish to thank Maleknaz Nayebi and Natalie Dean for helpful discussions on code architecture and workflow practices, respectively.

References

[AFG+04] Roy M Anderson, Christophe Fraser, Azra C Ghani, Christl A Donnelly, Steven Riley, Neil M Ferguson, Gabriel M Leung, Tai H Lam, and Anthony J Hedley. Epidemiology, transmission dynamics and control of sars: the 2002–2003 epidemic. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 359(1447):1091–1105, 2004. doi:10.1098/rstb.2004.1490.

[AJ09] Bashar Abdul-Jawad. *Groovy and Grails Recipes*. Springer, 2009.

[AUTE17] JJ Allaire, Kevin Ushey, Yuan Tang, and Dirk Eddelbuettel. *reticulate: R Interface to Python*, 2017. URL: https://github.com/rstudio/reticulate.

[BGB+18] Anna Bershteyn, Jaline Gerardin, Daniel Bridenbecker, Christopher W Lorton, Jonathan Bloedow, Robert S Baker, Guillaume Chabot-Couture, Ye Chen, Thomas Fischle, Kurt Frey, et al. Implementation and applications of EMOD, an individual-based multi-disease modeling platform. *Pathogens and disease*, 76(5):fty059, 2018. doi:10.1093/femspd/fty059.

[CHL+08] Myron S Cohen, Nick Hellmann, Jay A Levy, Kevin DeCock, Joep Lange, et al. The spread, treatment, and prevention of HIV-1: evolution of a global pandemic. *The Journal of Clinical Investigation*, 118(4):1244–1254, 2008. doi:10.1172/JCI34706.

[COSF20] Dennis L Chao, Assaf P Oron, Devabhaktuni Srikrishna, and Michael Famulare. Modeling layered non-pharmaceutical interventions against SARS-CoV-2 in the United States with Corvid. *MedRxiv*, 2020. doi:10.1101/2020.04.08.20058487.

[CSN+21] Jamie A Cohen, Robyn Margaret Stuart, Rafael C Nùñez, Katherine Rosenfeld, Bradley Wagner, Stewart Chang, Cliff Kerr, Michael Famulare, and Daniel J Klein. Mechanistic modeling of SARS-CoV-2 immune memory, variants, and vaccines. *medRxiv*, 2021. doi:10.1101/2021.05.31.21258018.

[Den20] Denim, Sue. Another Computer Simulation, Another Alarmist Prediction, 2020. URL: https://dailysceptic.org/schools-paper.

[Fam19] Mike Famulare. nCoV: preliminary estimates of the confirmed-case-fatality-ratio and infection-fatality-ratio, and initial pandemic risk assessment. *Institute for Disease Modeling*, 2019.

[Gar05] Laurie Garrett. The next pandemic. *Foreign Aff.*, 84:3, 2005. doi:10.2307/20034417.

[GP21] Brian E. Granger and Fernando Pérez. Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, 23(2):7–14, 2021. doi:10.1109/MCSE.2021.3059263.

[Hof20] Bert Hofman. The global pandemic. *Horizons: Journal of International Relations and Sustainable Development*, (16):60–69, 2020.

[HPN+21] Robert Hinch, William JM Probert, Anel Nurtay, Michelle Kendall, Chris Wymant, Matthew Hall, Katrina Lythgoe, Ana Bulas Cruz, Lele Zhao, Andrea Stewart, et al. OpenABM-Covid19—An agent-based model for non-pharmaceutical interventions against COVID-19 including contact tracing. *PLoS computational biology*, 17(7):e1009146, 2021. doi:10.1371/journal.pcbi.1009146.

[KAH+ng] Cliff C Kerr, Romesh G Abeysuriya, Vlad-Ștefan Harbuz, George L Chadderdon, Parham Saidi, Paula Sanz-Leon, James Jansson, Maria del Mar Quiroga, Sherrie Hughes, Rowan Martin-and Kelly, Jamie Cohen, Robyn M Stuart, and Anna Nachesa. Sciris: a Python library to simplify scientific computing. Available at http://paper.sciris.org, 2022 (forthcoming).

[KAK+19] David J Kedziora, Romesh Abeysuriya, Cliff C Kerr, George L Chadderdon, Vlad-Ștefan Harbuz, Sarah Metzger, David P Wilson, and Robyn M Stuart. The Cascade Analysis Tool: software to analyze and optimize care cascades. *Gates Open Research*, 3, 2019. doi:10.12688/gatesopenres.13031.2.

[KCP+20] Joel R Koo, Alex R Cook, Minah Park, Yinxiaohe Sun, Haoyang Sun, Jue Tao Lim, Clarence Tam, and Borame L Dickens. Interventions to mitigate early spread of sars-cov-2 in singapore: a modelling study. *The Lancet Infectious Diseases*, 20(6):678–688, 2020. doi:10.1016/S1473-3099(20)30162-6.

[KMS+21] Cliff C Kerr, Dina Mistry, Robyn M Stuart, Katherine Rosenfeld, Gregory R Hart, Rafael C Núñez, Jamie A Cohen, Prashanth Selvaraj, Romesh G Abeysuriya, Michał Jastrzębski, et al. Controlling COVID-19 via test-trace-quarantine. *Nature Communications*, 12(1):1–12, 2021. doi:10.1038/s41467-021-23276-9.

[KSM+21] Cliff C Kerr, Robyn M Stuart, Dina Mistry, Romesh G Abeysuriya, Katherine Rosenfeld, Gregory R Hart, Rafael C Núñez, Jamie A Cohen, Prashanth Selvaraj, Brittany Hagedorn, et al. Covasim: an agent-based model of COVID-19 dynamics and interventions. *PLOS Computational Biology*, 17(7):e1009149, 2021. doi:10.1371/journal.pcbi.1009149.

[LG+21] Junjiang Li, Philippe Giabbanelli, et al. Returning to a normal life via COVID-19 vaccines in the United States: a large-scale Agent-Based simulation study. *JMIR medical informatics*, 9(4):e27419, 2021. doi:10.2196/27419.

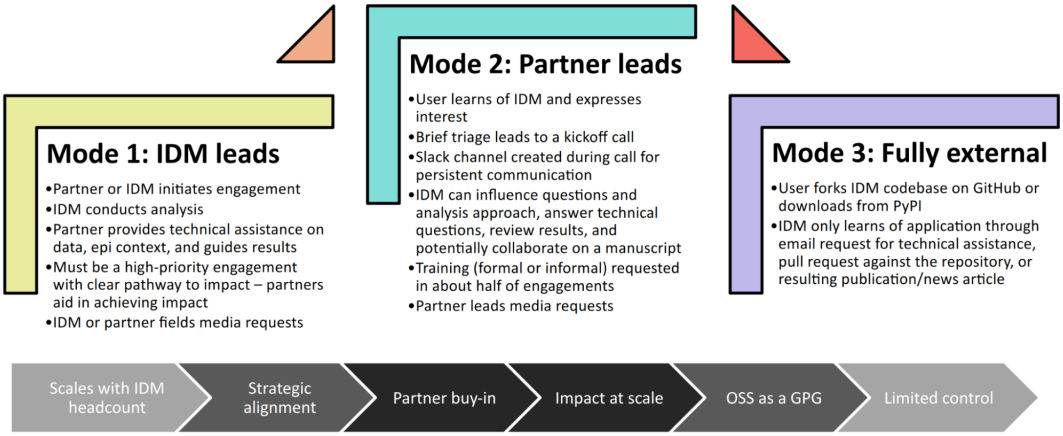


Fig. 8: *The three pathways to impact with Covasim, from high bandwidth/small scale to low bandwidth/large scale. IDM: Institute for Disease Modeling; OSS: open-source software; GPG: global public good; PyPI: Python Package Index.*

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015. doi:10.1145/2833157.2833162.

[Med20] The Lancet Respiratory Medicine. COVID-19: delay, mitigate, and communicate. *The Lancet Respiratory Medicine*, 8(4):321, 2020. doi:10.1016/S2213-2600(20)30128-4.

[OVCC+22] Michelle L O'Brien, Annie Valente, Guillaume Chabot-Couture, Joshua Proctor, Daniel Klein, Cliff Kerr, and Marita Zimmermann. FPSim: An agent-based model of family planning for informed policy decision-making. In *PAA 2022 Annual Meeting*. PAA, 2022.

[PeP22] PePy. PePy download statistics, 2022. URL: https://pepy.tech/project/covasim.

[PGKS+20] Jasmina Panovska-Griffiths, Cliff C Kerr, Robyn M Stuart, Dina Mistry, Daniel J Klein, Russell M Viner, and Chris Bonell. Determining the optimal strategy for reopening schools, the impact of test and trace interventions, and the risk of occurrence of a second COVID-19 epidemic wave in the UK: a modelling study. *The Lancet Child & Adolescent Health*, 4(11):817–827, 2020. doi:10.1016/S2352-4642(20)30250-9.

[PSN+21] Quang D Pham, Robyn M Stuart, Thuong V Nguyen, Quang C Luong, Quang D Tran, Thai Q Pham, Lan T Phan, Tan Q Dang, Duong N Tran, Hung T Do, et al. Estimating and mitigating the risk of COVID-19 epidemic rebound associated with reopening of international borders in Vietnam: a modelling study. *The Lancet Global Health*, 9(7):e916–e924, 2021. doi:10.1016/S2214-109X(21)00103-0.

[Qua19] QuantStack. *And voilá! Jupyter Blog*, 2019. URL: https://blog.jupyter.org/and-voil%C3%A0-f6a2c08a4a93.

[RSWS20] Joacim Rocklöv, Henrik Sjödin, and Annelies Wilder-Smith. COVID-19 outbreak on the Diamond Princess cruise ship: estimating the epidemic potential and effectiveness of public health countermeasures. *Journal of Travel Medicine*, 27(3):taaa030, 2020. doi:10.1093/jtm/taaa030.

[SAK+21] Robyn M Stuart, Romesh G Abeysuriya, Cliff C Kerr, Dina Mistry, Dan J Klein, Richard T Gray, Margaret Hellard, and Nick Scott. Role of masks, testing and contact tracing in preventing COVID-19 resurgences: a case study from New South Wales, Australia. *BMJ open*, 11(4):e045941, 2021. doi:10.1136/bmjopen-2020-045941.

[SHK16] Patrick R Saunders-Hastings and Daniel Krewski. Reviewing the history of pandemic influenza: understanding patterns of emergence and transmission. *Pathogens*, 5(4):66, 2016. doi:10.3390/pathogens5040066.

[SLSS+22] Paula Sanz-Leon, Nathan J Stevenson, Robyn M Stuart, Romesh G Abeysuriya, James C Pang, Stephen B Lambert, Cliff C Kerr, and James A Roberts. Risk of sustained SARS-CoV-2 transmission in Queensland, Australia. *Scientific reports*, 12(1):1–9, 2022. doi:10.1101/2021.06.08.21258599.

[SWC+22] Prashanth Selvaraj, Bradley G Wagner, Dennis L Chao, Maïna L'Azou Jackson, J Gabrielle Breugelmans, Nicholas Jackson, and Stewart T Chang. Rural prioritization may increase the impact of COVID-19 vaccines in a representative COVAX AMC country setting due to ongoing internal migration: A modeling study. *PLOS Global Public Health*, 2(1):e0000053, 2022. doi:10.1371/journal.pgph.0000053.

[Tea14] WHO Ebola Response Team. Ebola virus disease in west africa—the first 9 months of the epidemic and forward projections. *New England Journal of Medicine*, 371(16):1481–1495, 2014. doi:10.1056/NEJMoa1411100.

Pylira: deconvolution of images in the presence of Poisson noise

Axel Donath[‡*], Aneta Siemiginowska[‡], Vinay Kashyap[‡], Douglas Burke[‡], Karthik Reddy Solipuram[§], David van Dyk[¶]

Abstract—All physical and astronomical imaging observations are degraded by the finite angular resolution of the camera and telescope systems. The recovery of the true image is limited by both how well the instrument characteristics are known and by the magnitude of measurement noise. In the case of a high signal to noise ratio data, the image can be sharpened or “deconvolved” robustly by using established standard methods such as the Richardson-Lucy method. However, the situation changes for sparse data and the low signal to noise regime, such as those frequently encountered in X-ray and gamma-ray astronomy, where deconvolution leads inevitably to an amplification of noise and poorly reconstructed images. However, the results in this regime can be improved by making use of physically meaningful prior assumptions and statistically principled modeling techniques. One proposed method is the LIRA algorithm, which requires smoothness of the reconstructed image at multiple scales. In this contribution, we introduce a new python package called *Pylira*, which exposes the original C implementation of the LIRA algorithm to Python users. We briefly describe the package structure, development setup and show a Chandra as well as Fermi-LAT analysis example.

Index Terms—deconvolution, point spread function, poisson, low counts, X-ray, gamma-ray

Introduction

Any physical and astronomical imaging process is affected by the limited angular resolution of the instrument or telescope. In addition, the quality of the resulting image is also degraded by background or instrumental measurement noise and non-uniform exposure. For short wavelengths and associated low intensities of the signal, the imaging process consists of recording individual photons (often called "events") originating from a source of interest. This imaging process is typical for X-ray and gamma-ray telescopes, but images taken by magnetic resonance imaging or fluorescence microscopy show Poisson noise too. For each individual photon, the incident direction, energy and arrival time is measured. Based on this information, the event can be binned into two dimensional data structures to form an actual image.

As a consequence of the low intensities associated to the recording of individual events, the measured signal follows Poisson statistics. This imposes a non-linear relationship between the measured signal and true underlying intensity as well as a coupling of the signal intensity to the signal variance. Any statistically correct post-processing or reconstruction method thus requires a careful treatment of the Poisson nature of the measured image.

To maximise the scientific use of the data, it is often desired to correct the degradation introduced by the imaging process. Besides correction for non-uniform exposure and background noise this also includes the correction for the "blurring" introduced by the point spread function (PSF) of the instrument. Where the latter process is often called "deconvolution". Depending on whether the PSF of the instrument is known or not, one distinguishes between the "blind deconvolution" and "non blind deconvolution" process. For astronomical observations, the PSF can often either be simulated, given a model of the telescope and detector, or inferred directly from the data by observing far distant objects, which appear as a point source to the instrument.

While in other branches of astronomy deconvolution methods are already part of the standard analysis, such as the CLEAN algorithm for radio data, developed by [Hog74], this is not the case for X-ray and gamma-ray astronomy. As any deconvolution method aims to enhance small-scale structures in an image, it becomes increasingly hard to solve for the regime of low signal-to-noise ratio, where small-scale structures are more affected by noise.

The Deconvolution Problem

Basic Statistical Model

Assuming the data in each pixel d_i in the recorded counts image follows a Poisson distribution, the total likelihood of obtaining the measured image from a model image of the expected counts λ_i with N pixels is given by:

$$\mathscr{L}(\mathbf{d}|\lambda) = \prod_i^N \frac{\exp -d_i \lambda_i^{d_i}}{d_i!} \tag{1}$$

By taking the logarithm, dropping the constant terms and inverting the sign one can transform the product into a sum over pixels, which is also often called the *Cash* [Cas79] fit statistics:

$$\mathscr{C}(\lambda|\mathbf{d}) = \sum_i^N (\lambda_i - d_i \log \lambda_i) \tag{2}$$

Where the expected counts λ_i are given by the convolution of the true underlying flux distribution x_i with the PSF p_k:

$$\lambda_i = \sum_k x_i p_{i-k} \tag{3}$$

This operation is often called "forward modelling" or "forward folding" with the instrument response.

* Corresponding author: axel.donath@cfa.harvard.edu
‡ Center for Astrophysics | Harvard & Smithsonian
§ University of Maryland Baltimore County
¶ Imperial College London

Richardson Lucy (RL)

To obtain the most likely value of $\mathbf{x}_n$ given the data, one searches a maximum of the total likelihood function, or equivalently a of minimum $\mathscr{C}$. This high dimensional optimization problem can e.g., be solved by a classic gradient descent approach. Assuming the pixels values x_i of the true image as independent parameters, one can take the derivative of Eq. 2 with respect to the individual x_i. This way one obtains a rule for how to update the current set of pixels $\mathbf{x}_n$ in each iteration of the optimization:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \cdot \frac{\partial \mathscr{C}\,(\mathbf{d}|\mathbf{x})}{\partial x_i} \tag{4}$$

Where α is a factor to define the step size. This method is in general equivalent to the gradient descent and backpropagation methods used in modern machine learning techniques. This basic principle of solving the deconvolution problem for images with Poisson noise was proposed by [Ric72] and [Luc74]. Their method, named after the original authors, is often known as the *Richardson & Lucy* (RL) method. It was shown by [Ric72] that this converges to a maximum likelihood solution of Eq. 2. A Python implementation of the standard RL method is available e.g. in the *Scikit-Image* package [vdWSN+14].

Instead of the iterative, gradient descent based optimization it is also possible to sample from the posterior distribution using a simple Metropolis-Hastings [Has70] approach and uniform prior. This is demonstrated in one of the *Pylira* online tutorials (Introduction to Deconvolution using MCMC Methods).

RL Reconstruction Quality

While technically the RL method converges to a maximum likelihood solution, it mostly still results in poorly restored images, especially if extended emission regions are present in the image. The problem is illustrated in Fig. 1 using a simulated example image. While for a low number of iterations, the RL method still results in a smooth intensity distribution, the structure of the image decomposes more and more into a set of point-like sources with growing number of iterations.

Because of the PSF convolution, an extended emission region can decompose into multiple nearby point sources and still lead to good model prediction, when compared with the data. Those almost equally good solutions correspond to many narrow local minima or "spikes" in the global likelihood surface. Depending on the start estimate for the reconstructed image $\mathbf{x}$ the RL method will follow the steepest gradient and converge towards the nearest narrow local minimum. This problem has been described by multiple authors, such as [PR94] and [FBPW95].

Multi-Scale Prior & LIRA

One solution to this problem was described in [ECKvD04] and [CSv+11]. First, the simple forward folded model described in Eq. 3 can be extended by taking into account the non-uniform exposure e_i and an additional known background component b_i:

$$\lambda_i = \sum_k \left(e_i \cdot (x_i + b_i)\right) p_{i-k} \tag{5}$$

The background b_i can be more generally understood as a "baseline" image and thus include known structures, which are not of interest for the deconvolution process. E.g., a bright point source to model the core of an AGN while studying its jets.

Second, the authors proposed to extend the Poisson log-likelihood function (Equation 2) by a log-prior term that controls

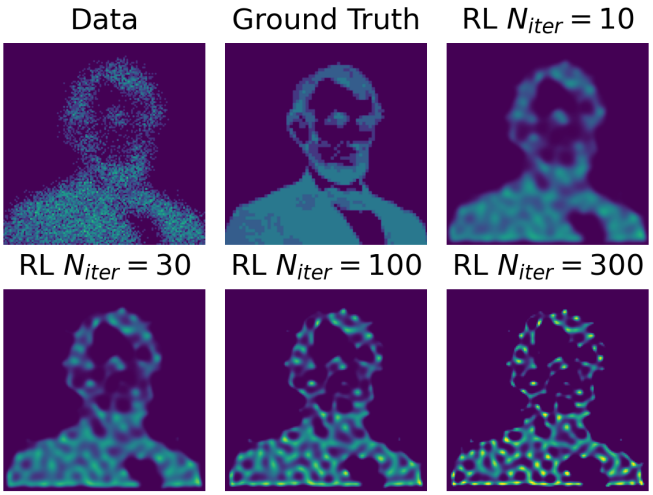


Fig. 1: *The images show the result of the RL algorithm applied to a simulated example dataset with varying numbers of iterations. The image in the upper left shows the simulated counts. Those have been derived from the ground truth (upper mid) by convolving with a Gaussian PSF of width $\sigma = 3$ pix and applying Poisson noise to it. The illustration uses the implementation of the RL algorithm from the Scikit-Image package [vdWSN+14].*

the smoothness of the reconstructed image on multiple spatial scales. Starting from the full resolution, the image pixels x_i are collected into 2 by 2 groups Q_k. The four pixel values associated with each group are divided by their sum to obtain a grid of "split proportions" with respect to the image down-sized by a factor of two along both axes. This process is repeated using the down sized image with pixel values equal to the sums over the 2 by 2 groups from the full-resolution image, and the process continues until the resolution of the image is only a single pixel, containing the total sum of the full-resolution image. This multi-scale representation is illustrated in Fig. 2.

For each of the 2x2 groups of the re-normalized images a Dirichlet distribution is introduced as a prior:

$$\phi_k \propto \text{Dirichlet}(\alpha_k, \alpha_k, \alpha_k, \alpha_k) \tag{6}$$

and multiplied across all 2x2 groups and resolution levels k. For each resolution level a smoothing parameter α_k is introduced. These hyper-parameters can be interpreted as having an information content equivalent of adding α_k "hallucinated" counts in each grouping. This effectively results in a smoothing of the image at the given resolution level. The distribution of α values at each resolution level is the further described by a hyper-prior distribution:

$$p(\alpha_k) = \exp\left(-\delta\alpha^3/3\right) \tag{7}$$

Resulting in a fully hierarchical Bayesian model. A more complete and detailed description of the prior definition is given in [ECKvD04].

The problem is then solved by using a Gibbs MCMC sampling approach. After a "burn-in" phase the sampling process typically reaches convergence and starts sampling from the posterior distribution. The reconstructed image is then computed as the mean of the posterior samples. As for each pixel a full distribution of its values is available, the information can also be used to compute the associated error of the reconstructed value. This is another main advantage over RL or Maxium A-Postori (MAP) algorithms.

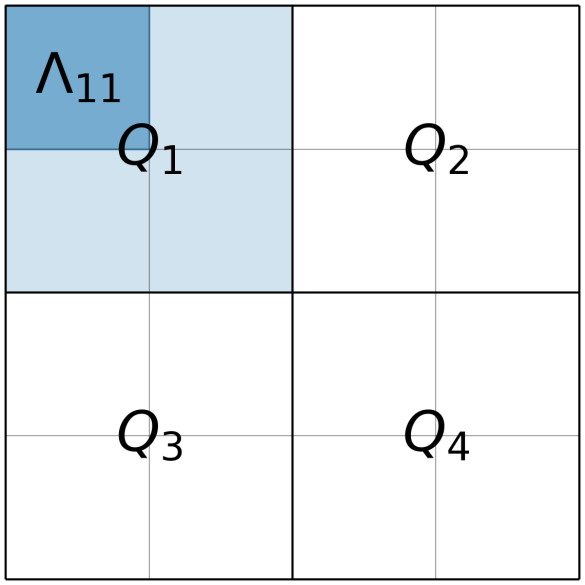


Fig. 2: *The image illustrates the multi-scale decomposition used in the LIRA prior for a 4x4 pixels example image. Each quadrant of 2x2 sub-images is labelled with Q_N. The sub-pixels in each quadrant are labelled Λ_{ij}. .*

The Pylira Package

Dependencies & Development

The *Pylira* package is a thin Python wrapper around the original *LIRA* implementation provided by the authors of [CSv+11]. The original algorithm was implemented in *C* and made available as a package for the *R Language* [R C20]. Thus the implementation depends on the *RMath* library, which is still a required dependency of *Pylira*. The Python wrapper was built using the *Pybind11* [JRM17] package, which allows to reduce the code overhead introduced by the wrapper to a minimum. For the data handling, *Pylira* relies on *Numpy* [HMvdW+20] arrays for the serialisation to the *FITS* data format on *Astropy* [Col18]. The (interactive) plotting functionality is achieved via *Matplotlib* [Hun07] and *Ipywidgets* [wc15], which are both optional dependencies. *Pylira* is openly developed on Github at https://github.com/astrostat/pylira. It relies on *GitHub Actions* as a continuous integration service and uses the *Read the Docs* service to build and deploy the documentation. The online documentation can be found on https://pylira.readthedocs.io. *Pylira* implements a set of unit tests to assure compatibility and reproducibility of the results with different versions of the dependencies and across different platforms. As *Pylira* relies on random sampling for the MCMC process an exact reproducibility of results is hard to achieve on different platforms; however the agreement of results is at least guaranteed in the statistical limit of drawing many samples.

Installation

Pylira is available via the Python package index (pypi.org), currently at version 0.1. As *Pylira* still depends on the *RMath* library, it is required to install this first. So the recommended way to install Pylira is on *MacOS* is:

```
$ brew install r
$ pip install pylira
```

On *Linux* the *RMath* dependency can be installed using standard package managers. For example on Ubuntu, one would do

```
$ sudo apt-get install r-base-dev r-base r-mathlib
$ pip install pylira
```

For more detailed instructions see Pylira installation instructions.

API & Subpackages

Pylira is structured in multiple sub-packages. The `pylira.src` module contains the original C implementation and the *Pybind11* wrapper code. The `pylira.core` sub-package contains the main Python API, `pylira.utils` includes utility functions for plotting and serialisation. And `pylira.data` implements multiple pre-defined datasets for testing and tutorials.

Analysis Examples

Simple Point Source

Pylira was designed to offer a simple Python class based user interface, which allows for a short learning curve of using the package for users who are familiar with Python in general and more specifically with *Numpy*. A typical complete usage example of the *Pylira* package is shown in the following:

```
import numpy as np
from pylira import LIRADeconvolver
from pylira.data import point_source_gauss_psf

# create example dataset
data = point_source_gauss_psf()

# define initial flux image
data["flux_init"] = data["flux"]

deconvolve = LIRADeconvolver(
    n_iter_max=3_000,
    n_burn_in=500,
    alpha_init=np.ones(5)
)

result = deconvolve.run(data=data)

# plot pixel traces, result shown in Figure 3
result.plot_pixel_traces_region(
    center_pix=(16, 16), radius_pix=3
)

# plot pixel traces, result shown in Figure 4
result.plot_parameter_traces()

# finally serialise the result
result.write("result.fits")
```

The main interface is exposed via the `LIRADeconvolver` class, which takes the configuration of the algorithm on initialisation. Typical configuration parameters include the total number of iterations `n_iter_max` and the number of "burn-in" iterations, to be excluded from the posterior mean computation. The data, represented by a simple Python `dict` data structure, contains a `"counts"`, `"psf"` and optionally `"exposure"` and `"background"` array. The dataset is then passed to the `LIRADeconvolver.run()` method to execute the deconvolution. The result is a `LIRADeconvolverResult` object, which features the possibility to write the result as a *FITS* file, as well as to inspect the result with diagnostic plots. The result of the computation is shown in the left panel of Fig. 3.

Diagnostic Plots

To validate the quality of the results *Pylira* provides many built-in diagnostic plots. One of these diagnostic plot is shown in the right panel of Fig. 3. The plot shows the image sampling trace

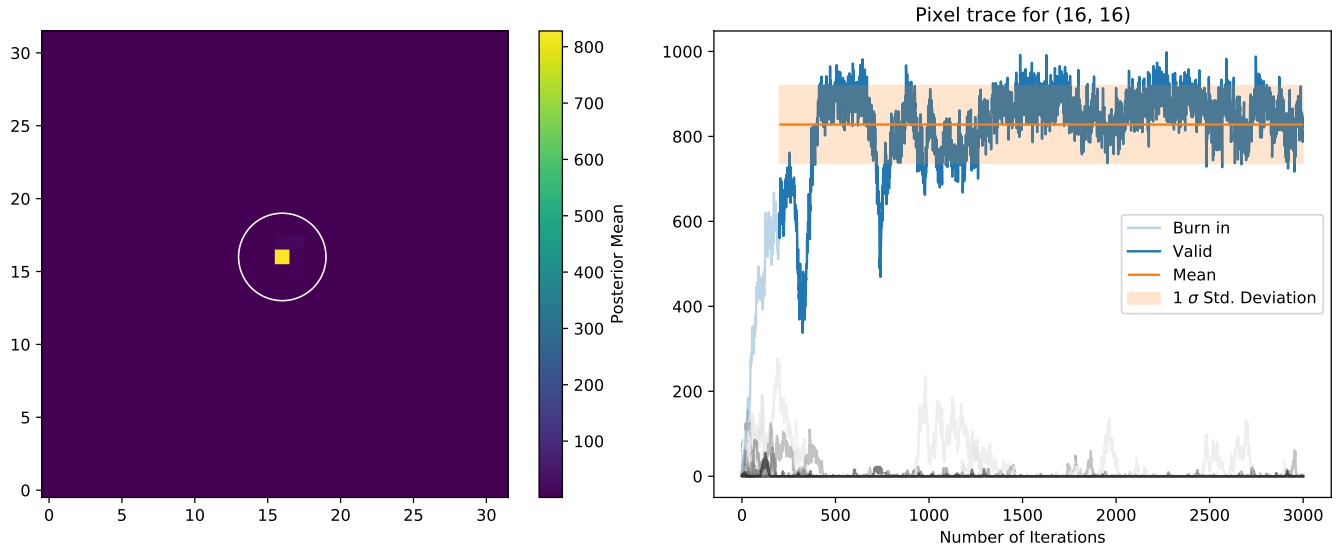


Fig. 3: *The curves show the traces of value the pixel of interest for a simulated point source and its neighboring pixels (see code example). The image on the left shows the posterior mean. The white circle in the image shows the circular region defining the neighboring pixels. The blue line on the right plot shows the trace of the pixel of interest. The solid horizontal orange line shows the mean value (excluding burn-in) of the pixel across all iterations and the shaded orange area the* $1\ \sigma$ *error region. The burn in phase is shown in transparent blue and ignored while computing the mean. The shaded gray lines show the traces of the neighboring pixels.*

for a single pixel of interest and its surrounding circular region of interest. This visualisation allows the user to assess the stability of a small region in the image e.g. an astronomical point source during the MCMC sampling process. Due to the correlation with neighbouring pixels, the actual value of a pixel might vary in the sampling process, which appears as "dips" in the trace of the pixel of interest and anti-correlated "peaks" in the one or mutiple of the surrounding pixels. In the example a stable state of the pixels of interest is reached after approximately 1000 iterations. This suggests that the number of burn-in iterations, which was defined beforehand, should be increased.

Pylira relies on an MCMC sampling approach to sample a series of reconstructed images from the posterior likelihood defined by Eq. 2. Along with the sampling, it marginalises over the smoothing hyper-parameters and optimizes them in the same process. To diagnose the validity of the results it is important to visualise the sampling traces of both the sampled images as well as hyper-parameters.

Figure 4 shows another typical diagnostic plot created by the code example above. In a multi-panel figure, the user can inspect the traces of the total log-posterior as well as the traces of the smoothing parameters. Each panel corresponds to the smoothing hyper parameter introduced for each level of the multi-scale representation of the reconstructed image. The figure also shows the mean value along with the $1\ \sigma$ error region. In this case, the algorithm shows stable convergence after a burn-in phase of approximately 200 iterations for the log-posterior as well as all of the multi-scale smoothing parameters.

Astronomical Analysis Examples

Both in the X-ray as well as in the gamma-ray regime, the Galactic Center is a complex emission region. It shows point sources, extended sources, as well as underlying diffuse emission and thus represents a challenge for any astronomical data analysis.

Chandra is a space-based X-ray observatory, which has been in operation since 1999. It consists of nested cylindrical paraboloid and hyperboloid surfaces, which form an imaging optical system for X-rays. In the focal plane, it has multiple instruments for different scientific purposes. This includes a high-resolution camera (HRC) and an Advanced CCD Imaging Spectrometer (ACIS). The typical angular resolution is 0.5 arcsecond and the covered energy ranges from 0.1 - 10 keV.

Figure 5 shows the result of the *Pylira* algorithm applied to Chandra data of the Galactic Center region between 0.5 and 7 keV. The PSF was obtained from simulations using the *simulate_psf* tool from the official Chandra science tools *ciao 4.14* [FMA+06]. The algorithm achieves both an improved spatial resolution as well as a reduced noise level and higher contrast of the image in the right panel compared to the unprocessed counts data shown in the left panel.

As a second example, we use data from the Fermi Large Area Telescope (LAT). The Fermi-LAT is a satellite-based imaging gamma-ray detector, which covers an energy range of 20 MeV to >300 GeV. The angular resolution varies strongly with energy and ranges from 0.1 to >10 degree[1].

Figure 6 shows the result of the *Pylira* algorithm applied to Fermi-LAT data above 1 GeV to the region around the Galactic Center. The PSF was obtained from simulations using the *gtpsf* tool from the official *Fermitools v2.0.19* [Fer19]. First, one can see that the algorithm achieves again a considerable improvement in the spatial resolution compared to the raw counts. It clearly resolves multiple point sources left to the bright Galactic Center source.

Summary & Outlook

The *Pylira* package provides Python wrappers for the LIRA algorithm. It allows the deconvolution of low-counts data following

1. https://www.slac.stanford.edu/exp/glast/groups/canda/lat_Performance.htm

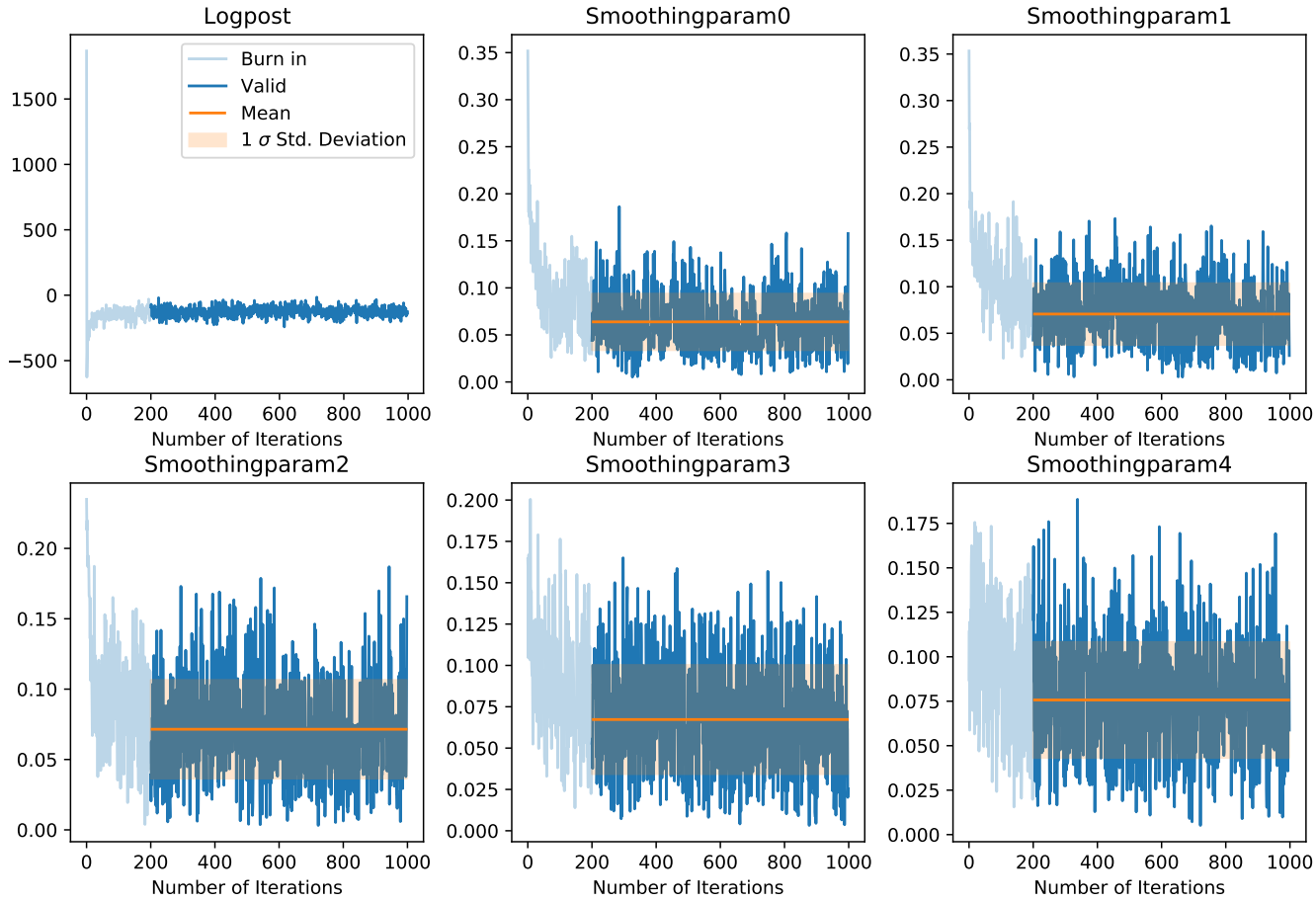


Fig. 4: *The curves show the traces of the log posterior value as well as traces of the values of the prior parameter values. The* SmoothingparamN *parameters correspond to the smoothing parameters* α_N *per multi-scale level. The solid horizontal orange lines show the mean value, the shaded orange area the* 1 σ *error region. The burn in phase is shown transparent and ignored while estimating the mean.*

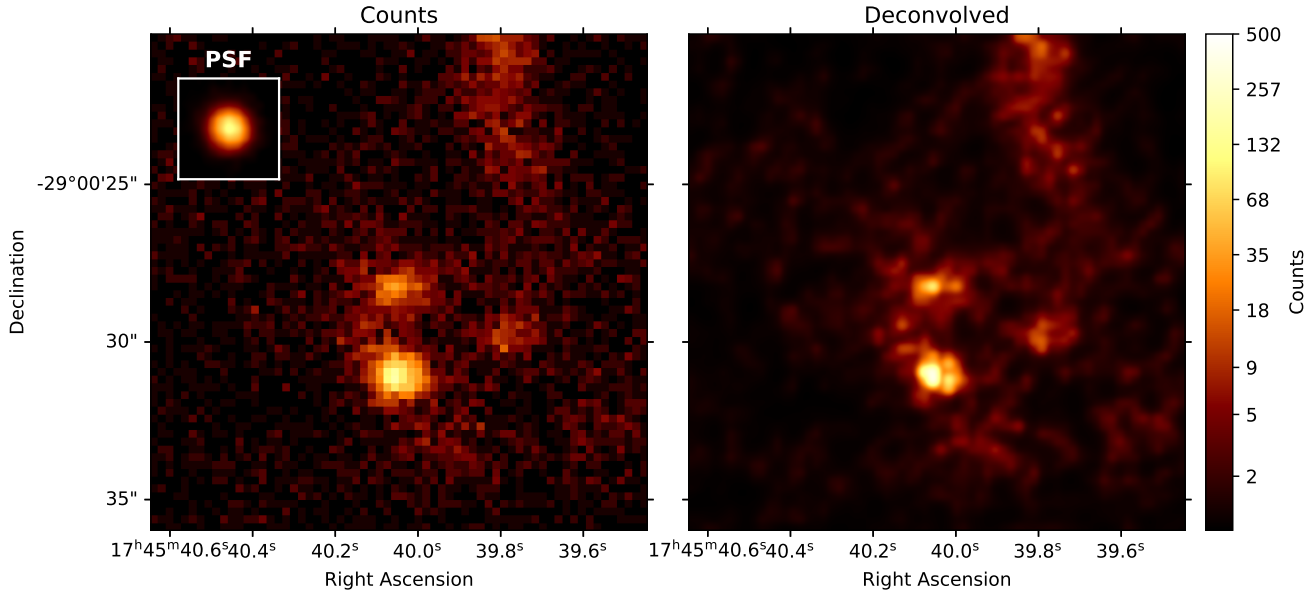


Fig. 5: *Pylira applied to Chandra ACIS data of the Galactic Center region, using the observation IDs* 4684 *and* 4684*. The image on the left shows the raw observed counts between 0.5 and 7 keV. The image on the right shows the deconvolved version. The LIRA hyperprior values were chosen as* ms_al_kap1=1, ms_al_kap2=0.02, ms_al_kap3=1. *No baseline background model was included.*

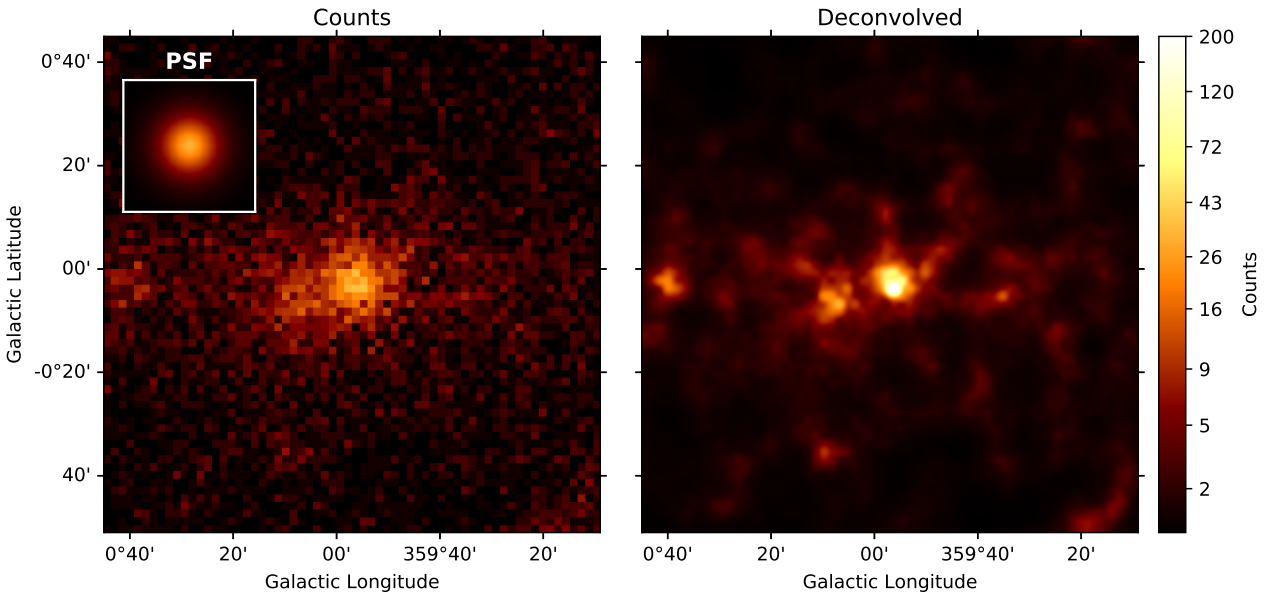


Fig. 6: *Pylira applied to Fermi-LAT data from the Galactic Center region. The image on the left shows the raw measured counts between 5 and 1000 GeV. The image on the right shows the deconvolved version. The LIRA hyperprior values were chosen as* ms_al_kap1=1, ms_al_kap2=0.02, ms_al_kap3=1. *No baseline background model was included.*

Poisson statistics using a Bayesian sampling approach and a multi-scale smoothing prior assumption. The results can be easily written to FITS files and inspected by plotting the trace of the sampling process. This allows users to check for general convergence as well as pixel to pixel correlations for selected regions of interest. The package is openly developed on GitHub and includes tests and documentation, such that it can be maintained and improved in the future, while ensuring consistency of the results. It comes with multiple built-in test datasets and explanatory tutorials in the form of Jupyter notebooks. Future plans include the support for parallelisation or distributed computing, more flexible prior definitions and the possibility to account for systematic errors on the PSF during the sampling process.

Acknowledgements

This work was conducted under the auspices of the CHASC International Astrostatistics Center. CHASC is supported by NSF grants DMS-21-13615, DMS-21-13397, and DMS-21-13605; by the UK Engineering and Physical Sciences Research Council [EP/W015080/1]; and by NASA 18-APRA18-0019. We thank CHASC members for many helpful discussions, especially Xiao-Li Meng and Katy McKeough. DvD was also supported in part by a Marie-Skodowska-Curie RISE Grant (H2020-MSCA-RISE-2019-873089) provided by the European Commission. Aneta Siemiginowska, Vinay Kashyap, and Doug Burke further acknowledge support from NASA contract to the Chandra X-ray Center NAS8-03060.

References

[Cas79] W. Cash. Parameter estimation in astronomy through application of the likelihood ratio. *The Astrophysical Journal*, 228:939–947, March 1979. doi:10.1086/156922.

[Col18] Astropy Collaboration. The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *The Astrophysical Journal*, 156(3):123, September 2018. arXiv:1801.02634, doi:10.3847/1538-3881/aabc4f.

[CSv+11] A. Connors, N. M. Stein, D. van Dyk, V. Kashyap, and A. Siemiginowska. LIRA — The Low-Counts Image Restoration and Analysis Package: A Teaching Version via R. In I. N. Evans, A. Accomazzi, D. J. Mink, and A. H. Rots, editors, *Astronomical Data Analysis Software and Systems XX*, volume 442 of *Astronomical Society of the Pacific Conference Series*, page 463, July 2011.

[ECKvD04] David N. Esch, Alanna Connors, Margarita Karovska, and David A. van Dyk. An image restoration technique with error estimates. *The Astrophysical Journal*, 610(2):1213–1227, aug 2004. URL: https://doi.org/10.1086/421761, doi:10.1086/421761.

[FBPW95] D. A. Fish, A. M. Brinicombe, E. R. Pike, and J. G. Walker. Blind deconvolution by means of the richardson–lucy algorithm. *J. Opt. Soc. Am. A*, 12(1):58–65, Jan 1995. URL: http://opg.optica.org/josaa/abstract.cfm?URI=josaa-12-1-58, doi:10.1364/JOSAA.12.000058.

[Fer19] Fermi Science Support Development Team. Fermitools: Fermi Science Tools. Astrophysics Source Code Library, record ascl:1905.011, May 2019. arXiv:1905.011.

[FMA+06] Antonella Fruscione, Jonathan C. McDowell, Glenn E. Allen, Nancy S. Brickhouse, Douglas J. Burke, John E. Davis, Nick Durham, Martin Elvis, Elizabeth C. Galle, Daniel E. Harris, David P. Huenemoerder, John C. Houck, Bish Ishibashi, Margarita Karovska, Fabrizio Nicastro, Michael S. Noble, Michael A. Nowak, Frank A. Primini, Aneta Siemiginowska, Randall K. Smith, and Michael Wise. CIAO: Chandra's data analysis system. In David R. Silva and Rodger E. Doxsey, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6270 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, page 62701V, June 2006. doi:10.1117/12.671760.

[Has70] W. K. Hastings. Monte Carlo Sampling Methods using Markov Chains and their Applications. *Biometrika*, 57(1):97–109, April 1970. doi:10.1093/biomet/57.1.97.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[Hog74] J. A. Hogbom. Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines. *Astronomy and Astrophysics Supplement*, 15:417, June 1974.

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

[JRM17] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[Luc74] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 79:745, June 1974. doi:10.1086/111605.

[PR94] K. M. Perry and S. J. Reeves. Generalized Cross-Validation as a Stopping Rule for the Richardson-Lucy Algorithm. In Robert J. Hanisch and Richard L. White, editors, *The Restoration of HST Images and Spectra - II*, page 97, January 1994. doi:10.1002/ima.1850060412.

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020. URL: https://www.R-project.org/.

[Ric72] William Hadley Richardson. Bayesian-Based Iterative Method of Image Restoration. *Journal of the Optical Society of America (1917-1983)*, 62(1):55, January 1972. doi:10.1364/josa.62.000055.

[vdWSN+14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014. URL: https://doi.org/10.7717/peerj.453, doi:10.7717/peerj.453.

[wc15] Jupyter widgets community. ipywidgets, a github repository. Retrieved from https://github.com/jupyter-widgets/ipywidgets, 2015.

Codebraid Preview for VS Code: Pandoc Markdown Preview with Jupyter Kernels

Geoffrey M. Poore[‡*]

Abstract—Codebraid Preview is a VS Code extension that provides a live preview of Pandoc Markdown documents with optional support for executing embedded code. Unlike typical Markdown previews, all Pandoc features are fully supported because Pandoc itself generates the preview. The Markdown source and the preview are fully integrated with features like bidirectional scroll sync. The preview supports LaTeX math via KaTeX. Code blocks and inline code can be executed with Codebraid, using either its built-in execution system or Jupyter kernels. For executed code, any combination of the code and its output can be displayed in the preview as well as the final document. Code execution is non-blocking, so the preview always remains live and up-to-date even while code is still running.

Index Terms—reproducibility, dynamic report generation, literate programming, Python, Pandoc, Markdown, Project Jupyter

Introduction

Pandoc [JM22] is increasingly a foundational tool for creating scientific and technical documents. It provides Pandoc's Markdown and other Markdown variants that add critical features absent in basic Markdown, such as citations, footnotes, mathematics, and tables. At the same time, Pandoc simplifies document creation by providing conversion from Markdown (and other formats) to formats like LaTeX, HTML, Microsoft Word, and PowerPoint. Pandoc is especially useful for documents with embedded code that is executed during the build process. RStudio's RMarkdown [RSt20] and more recently Quarto [RSt22] leverage Pandoc to convert Markdown documents to other formats, with code execution provided by knitr [YX15]. JupyterLab [GP21] centers the writing experience around an interactive, browser-based notebook instead of a Markdown document, but still relies on Pandoc for export to formats other than HTML [Jup22]. There are also ways to interact with a Jupyter Notebook as a Markdown document, such as Jupytext [MWtJT20] and Pandoc's own native Jupyter support.

Writing with Pandoc's Markdown or a similar Markdown variant has advantages when multiple output formats are required, since Pandoc provides the conversion capabilities. Pandoc Markdown variants can also serve as a simpler syntax when creating HTML, LaTeX, or similar documents. They allow HTML and LaTeX to be intermixed with Markdown syntax. They also support including raw chunks of text in other formats such as reStructuredText. When executable code is involved, the RMarkdown-style approach of Markdown with embedded code can sometimes be more convenient than a browser-based Jupyter notebook since the writing process involves more direct interaction with the complete document source.

While using a Pandoc Markdown variant as a source format brings many advantages, the actual writing process itself can be less than ideal, especially when executable code is involved. Pandoc Markdown variants are so powerful precisely because they provide so many extensions to Markdown, but this also means that they can only be fully rendered by Pandoc itself. When text editors such as VS Code provide a built-in Markdown preview, typically only a small subset of Pandoc features is supported, so the representation of the document output will be inaccurate. Some editors provide a visual Markdown editing mode, in which a partially rendered version of the document is displayed in the editor and menus or keyboard shortcuts may replace the direct entry of Markdown syntax. These generally suffer from the same issue. This is only exacerbated when the document embeds code that is executed during the build process, since that goes even further beyond basic Markdown.

An alternative is to use Pandoc itself to generate HTML or PDF output, and then display this as a preview. Depending on the text editor used, the HTML or PDF might be displayed within the text editor in a panel beside the document source, or in a separate browser window or PDF viewer. For example, Quarto offers both possibilities, depending on whether RStudio, VS Code, or another editor is used.[1] While this approach resolves the inaccuracy issues of a basic Markdown preview, it also gives up features such as scroll sync that tightly integrate the Markdown source with the preview. In the case of executable code, there is the additional issue of a time delay in rendering the preview. Pandoc itself can typically convert even a relatively long document in under one second. However, when code is executed as part of the document build process, preview update is blocked until code execution completes.

This paper introduces Codebraid Preview, a VS Code extension that provides a live preview of Pandoc Markdown documents with optional support for executing embedded code. Codebraid Preview provides a Pandoc-based preview while avoiding most of the traditional drawbacks of this approach. The next section

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

1. The RStudio editor is unique in also offering a Pandoc-based visual editing mode, starting with version 1.4 from January 2021 (https://www.rstudio.com/blog/announcing-rstudio-1-4/).

provides an overview of features. This is followed by sections focusing on scroll sync, LaTeX support, and code execution as examples of solutions and remaining challenges in creating a better Pandoc writing experience.

Overview of Codebraid Preview

Codebraid Preview can be installed through the VS Code extension manager. Development is at https://github.com/gpoore/codebraid-preview-vscode. Pandoc must be installed separately (https://pandoc.org/). For code execution capabilities, Codebraid must also be installed (https://github.com/gpoore/codebraid).

The preview panel can be opened using the VS Code command palette, or by clicking the Codebraid Preview button that is visible when a Markdown document is open. The preview panel takes the document in its current state, converts it into HTML using Pandoc, and displays the result using a webview. An example is shown in Figure 1. Since the preview is generated by Pandoc, all Pandoc features are fully supported.

By default, the preview updates automatically whenever the Markdown source is changed. There is a short user-configurable minimum update interval. For shorter documents, sub-second updates are typical.

The preview uses the same styling CSS as VS Code's built-in Markdown preview, so it automatically adjusts to the VS Code color theme. For example, changing between light and dark themes changes the background and text colors in the preview.

Codebraid Preview leverages recent Pandoc advances to provide bidirectional scroll sync between the Markdown source and the preview for all CommonMark-based Markdown variants that Pandoc supports (`commonmark`, `gfm`, `commonmark_x`). By default, Codebraid Preview treats Markdown documents as `commonmark_x`, which is CommonMark with Pandoc extensions for features like math, footnotes, and special list types. The preview still works for other Markdown variants, but scroll sync is disabled. By default, scroll sync is fully bidirectional, so scrolling either the source or the preview will cause the other to scroll to the corresponding location. Scroll sync can instead be configured to be only from source to preview or only from preview to source. As far as I am aware, this is the first time that scroll sync has been implemented in a Pandoc-based preview.

The same underlying features that make scroll sync possible are also used to provide other preview capabilities. Double-clicking in the preview moves the cursor in the editor to the corresponding line of the Markdown source.

Since many Markdown variants support LaTeX math, the preview includes math support via KaTeX [EA22].

Codebraid Preview can simply be used for writing plain Pandoc documents. Optional execution of embedded code is possible with Codebraid [GMP19], using its built-in code execution system or Jupyter kernels. When Jupyter kernels are used, it is possible to obtain the same output that would be present in a Jupyter notebook, including rich output such as plots and mathematics. It is also possible to specify a custom display so that only a selected combination of code, stdout, stderr, and rich output is shown while the rest are hidden. Code execution is decoupled from the preview process, so the Markdown source can be edited and the preview can update even while code is running in the background. As far as I am aware, no previous software for executing code in Markdown has supported building a document with partial code output before execution has completed.

There is also support for document export with Pandoc, using the VS Code command palette or the export-with-Pandoc button.

Scroll sync

Tight source-preview integration requires a source map, or a mapping from characters in the source to characters in the output. Due to Pandoc's parsing algorithms, tracking source location during parsing is not possible in the general case.[2]

Pandoc 2.11.3 was released in December 2020. It added a `sourcepos` extension for CommonMark and formats based on it, including GitHub-Flavored Markdown (GFM) and `commonmark_x` (CommonMark plus extensions similar to Pandoc's Markdown). The CommonMark parser uses a different parsing algorithm from the Pandoc's Markdown parser, and this algorithm permits tracking source location. For the first time, it was possible to construct a source map for a Pandoc input format.

Codebraid Preview defaults to `commonmark_x` as an input format, since it provides the most features of all CommonMark-based formats. Features continue to be added to `commonmark_x` and it is gradually nearing feature parity with Pandoc's Markdown. Citations are perhaps the most important feature currently missing.[3]

Codebraid Preview provides full bidirectional scroll sync between source and preview for all CommonMark-based formats, using data provided by `sourcepos`. In the output HTML, the first image or inline text element created by each Markdown source line is given an id attribute corresponding to the source line number. When the source is scrolled to a given line range, the preview scrolls to the corresponding HTML elements using these id attributes. When the preview is scrolled, the visible HTML elements are detected via the Intersection Observer API.[4] Then their id attributes are used to determine the corresponding Markdown line range, and the source scrolls to those lines.

Scroll sync is slightly more complicated when working with output that is generated by executed code. For example, if a code block is executed and creates several plots in the preview, there isn't necessarily a way to trace each individual plot back to a particular line of code in the Markdown source. In such cases, the line range of the executed code is mapped proportionally to the vertical space occupied by its output.

Pandoc supports multi-file documents. It can be given a list of files to combine into a single output document. Codebraid Preview provides scroll sync for multi-file documents. For example, suppose a document is divided into two files in the same directory, `chapter_1.md` and `chapter_2.md`. Treating these as a single document involves creating a YAML configuration file `_codebraid_preview.yaml` that lists the files:

```
input-files:
- chapter_1.md
- chapter_2.md
```

Now launching a preview from either `chapter_1.md` or `chapter_2.md` will display a preview that combines both files. When the preview is scrolled, the editor scrolls to the corresponding source location, automatically switching between

2. See for example https://github.com/jgm/pandoc/issues/4565.

3. The Pandoc Roadmap at https://github.com/jgm/pandoc/wiki/Roadmap summarizes current `commonmark_x` capabilities.

4. For technical details, https://www.w3.org/TR/intersection-observer/. For an overview, https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API.

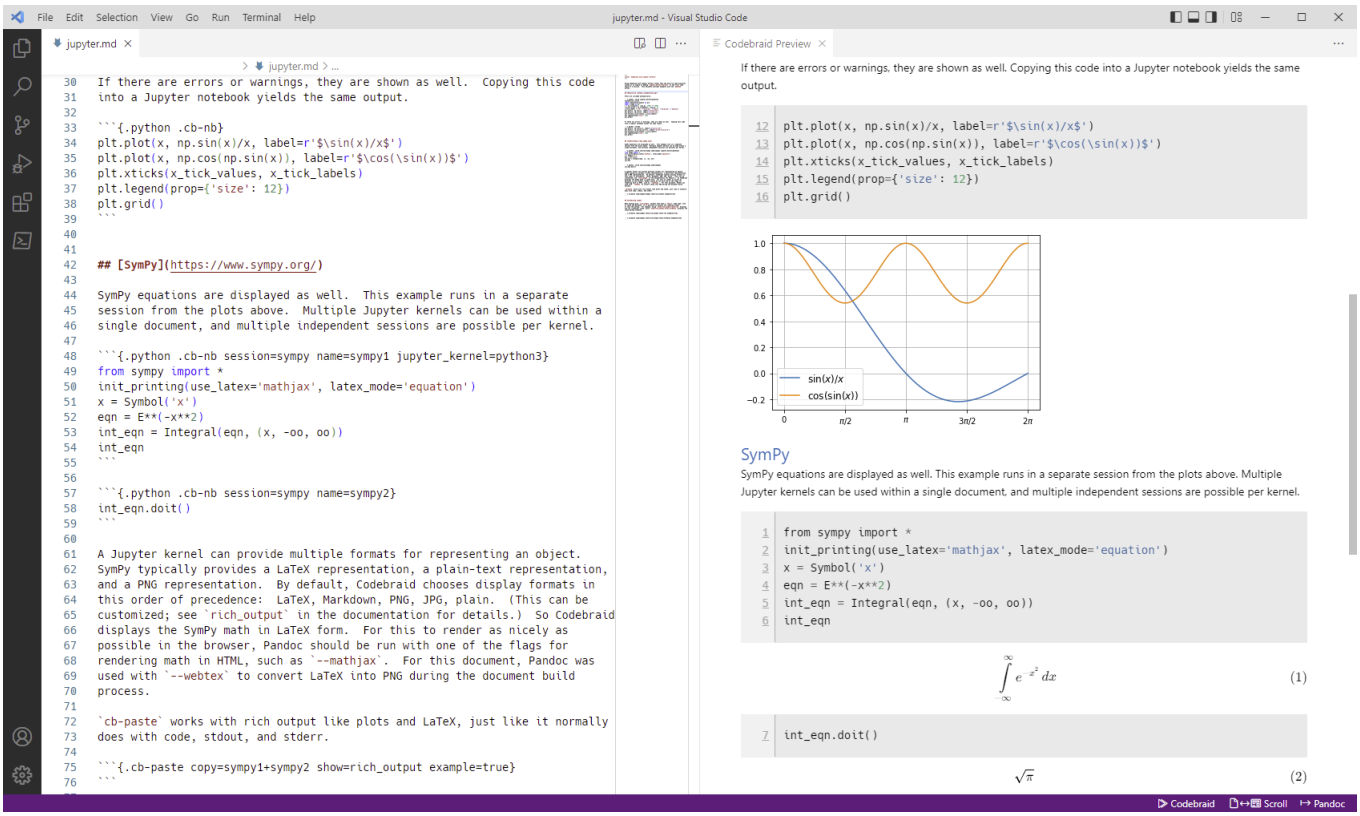


Fig. 1: *Screenshot of a Markdown document with Codebraid Preview in VS Code. This document uses Codebraid to execute code with Jupyter kernels, so all plots and math visible in the preview are generated during document build.*

`chapter_1.md` and `chapter_2.md` depending on the part of the preview that is visible.

The preview still works when the input format is set to a non-CommonMark format, but in that case scroll sync is disabled. If Pandoc adds `sourcepos` support for additional input formats in the future, scroll sync will work automatically once Codebraid Preview adds those formats to the supported list. It is possible to attempt to reconstruct a source map by performing a parallel string search on Pandoc output and the original source. This can be error-prone due to text manipulation during format conversion, but in the future it may be possible to construct a good enough source map to extend basic scroll sync support to additional input formats.

LaTeX support

Support for mathematics is one of the key features provided by many Markdown variants in Pandoc, including `commonmark_x`. Math support in the preview panel is supplied by KaTeX [EA22], which is a JavaScript library for rendering LaTeX math in the browser.

One of the disadvantages of using Pandoc to create the preview is that every update of the preview is a complete update. This makes the preview more sensitive to HTML rendering time. In contrast, in a Jupyter notebook, it is common to write Markdown in multiple cells which are rendered separately and independently.

MathJax [Mat22] provides a broader range of LaTeX support than KaTeX, and is used in software such as JupyterLab and Quarto. While MathJax performance has improved significantly since the release of version 3.0 in 2019, KaTeX can still have a speed advantage, so it is currently the default due to the importance of HTML rendering. In the future, optional MathJax support may be needed to provide broader math support. For some applications, it may also be worth considering caching pre-rendered or image versions of equations to improve performance.

Code execution

Optional support for executing code embedded in Markdown documents is provided by Codebraid [GMP19]. Codebraid uses Pandoc to convert a document into an abstract syntax tree (AST), then extracts any inline or block code marked with Codebraid attributes from the AST, executes the code, and finally formats the code output so that Pandoc can use it to create the final output document. Code execution is performed with Codebraid's own built-in system or with Jupyter kernels. For example, the code block

````
```{.python .cb-run}
print("Hello *world!*")
```
````

would result in

Hello *world!*

after processing by Codebraid and finally Pandoc. The `.cb-run` is a Codebraid attribute that marks the code block for execution and specifies the default display of code output. Further examples of Codebraid usage are visible in Figure 1.

Mixing a live preview with executable code provides potential usability and security challenges. By default, code only runs when the user selects execution in the VS Code command palette or clicks the Codebraid execute button. When the preview automatically updates as a result of Markdown source changes, it only uses

cached code output. Stale cached output is detected by hashing executed code, and then marked in the preview to alert the user.

The standard approach to executing code within Markdown documents blocks the document build process until all code has finished running. Code is extracted from the Markdown source and executed. Then the output is combined with the original source and passed on to Pandoc or another Markdown application for final conversion. This is the approach taken by RMarkdown, Quarto, and similar software, as well as by Codebraid until recently. This design works well for building a document a single time, but blocking until all code has executed is not ideal in the context of a document preview.

Codebraid now offers a new mode of code execution that allows a document to be rebuilt continuously during code execution, with each build including all code output available at that time. This process involves the following steps:

1) The user selects code execution. Codebraid Preview passes the document to Codebraid. Codebraid begins code execution.
2) As soon as any code output is available, Codebraid immediately streams this back to Codebraid Preview. The output is in a format compatible with the YAML metadata block at the start of Pandoc Markdown documents. The output includes a hash of the code that was executed, so that code changes can be detected later.
3) If the document is modified while code is running or if code output is received, Codebraid Preview rebuilds the preview. It creates a copy of the document with all current Codebraid output inserted into the YAML metadata block at the start of the document. This modified document is then passed to Pandoc. Pandoc runs with a Lua filter[5] that modifies the document AST before final conversion. The filter removes all code marked with Codebraid attributes from the AST, and replaces it with the corresponding code output stored in the AST metadata. If code has been modified since execution began, this is detected with the hash of the code, and an HTML class is added to the output that will mark it visually as stale output. Code that does not yet have output is replaced by a visible placeholder to indicate that code is still running. When the Lua filter finishes AST modifications, Pandoc completes the document build, and the preview updates.
4) As long as code is executing, the previous process repeats whenever the preview needs to be rebuilt.
5) Once code execution completes, the most recent output is reused for all subsequent preview updates until the next time the user chooses to execute code. Any code changes continue to be detected by hashing the code during the build process, so that the output can be marked visually as stale in the preview.

The overall result of this process is twofold. First, building a document involving executed code is nearly as fast as building a plain Pandoc document. The additional output metadata plus the filter are the only extra elements involved in the document build, and Pandoc Lua filters have excellent performance. Second, the output for each code chunk appears in the preview almost immediately after the chunk finishes execution.

5. For an overview of Lua filters, see https://pandoc.org/lua-filters.html.

While this build process is significantly more interactive than what has been possible previously, it also suggests additional avenues for future exploration. Codebraid's built-in code execution system is designed to execute a predefined sequence of code chunks and then exit. Jupyter kernels are currently used in the same manner to avoid any potential issues with out-of-order execution. However, Jupyter kernels can receive and execute code indefinitely, which is how they commonly function in Jupyter notebooks. Instead of starting a new Jupyter kernel at the beginning of each code execution cycle, it would be possible to keep the kernel from the previous execution cycle and only pass modified code chunks to it. This would allow the same out-of-order execution issues that are possible in a Jupyter notebook. Yet that would make possible much more rapid code output, particularly in cases where large datasets must be loaded or significant preprocessing is required.

Conclusion

Codebraid Preview represents a significant advance in tools for writing with Pandoc. For the first time, it is possible to preview a Pandoc Markdown document using Pandoc itself while having features like scroll sync between the Markdown source and the preview. When embedded code needs to be executed, it is possible to see code output in the preview and to continue editing the document during code execution, instead of having to wait until code finishes running.

Codebraid Preview or future previewers that follow this approach may be perfectly adequate for shorter and even some longer documents, but at some point a combination of document length, document complexity, and mathematical content will strain what is possible and ultimately decrease preview update frequency. Every update of the preview involves converting the entire document with Pandoc and then rendering the resulting HTML.

On the parsing side, Pandoc's move toward CommonMark-based Markdown variants may eventually lead to enough standardization that other implementations with the same syntax and features are possible. This in turn might enable entirely new approaches. An ideal scenario would be a Pandoc-compatible JavaScript-based parser that can parse multiple Markdown strings while treating them as having a shared document state for things like labels, references, and numbering. For example, this could allow Pandoc Markdown within a Jupyter notebook, with all Markdown content sharing a single document state, maybe with each Markdown cell being automatically updated based on Markdown changes elsewhere.

Perhaps more practically, on the preview display side, there may be ways to optimize how the HTML generated by Pandoc is loaded in the preview. A related consideration might be alternative preview formats. There is a significant tradition of tight source-preview integration in LaTeX (for example, [Lau08]). In principle, Pandoc's `sourcepos` extension should make possible Markdown to PDF synchronization, using LaTeX as an intermediary.

References

[EA22] Emily Eisenberg and Sophie Alpert. KaTeX: The fastest math typesetting library for the web, 2022. URL: https://katex.org/.

[GMP19] Geoffrey M. Poore. Codebraid: Live Code in Pandoc Markdown. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 54 – 61, 2019. doi:10.25080/Majora-7ddc1dd1-008.

[GP21] Brian E. Granger and Fernando Pérez. Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, 23(2):7–14, 2021. doi:10.1109/MCSE.2021.3059263.

[JM22] John MacFarlane. Pandoc: a universal document converter, 2006–2022. URL: https://pandoc.org/.

[Jup22] Jupyter Development Team. nbconvert: Convert Notebooks to other formats, 2015–2022. URL: https://nbconvert.readthedocs.io.

[Lau08] Jerôme Laurens. Direct and reverse synchronization with SyncTEX. *TUGBoat*, 29(3):365–371, 2008.

[Mat22] MathJax. MathJax: Beautiful and accessible math in all browsers, 2009–2022. URL: https://www.mathjax.org/.

[MWtJT20] Marc Wouts and the Jupytext Team. Jupyter notebooks as Markdown documents, Julia, Python or R scripts, 2018–2020. URL: https://jupytext.readthedocs.io/.

[RSt20] RStudio Inc. R Markdown, 2016–2020. URL: https://rmarkdown.rstudio.com/.

[RSt22] RStudio Inc. Welcome to Quarto, 2022. URL: https://quarto.org/.

[YX15] Yihui Xie. *Dynamic Documents with R and knitr*. Chapman & Hall/CRC Press, 2015.

Incorporating Task-Agnostic Information in Task-Based Active Learning Using a Variational Autoencoder

Curtis Godwin[‡†*], Meekail Zain[§†*], Nathan Safir[‡], Bella Humphrey[§], Shannon P Quinn[§¶]

Abstract—It is often much easier and less expensive to collect data than to label it. Active learning (AL) ([Set09]) responds to this issue by selecting which unlabeled data are best to label next. Standard approaches utilize task-aware AL, which identifies informative samples based on a trained supervised model. Task-agnostic AL ignores the task model and instead makes selections based on learned properties of the dataset. We seek to combine these approaches and measure the contribution of incorporating task-agnostic information into standard AL, with the suspicion that the extra information in the task-agnostic features may improve the selection process. We test this on various AL methods using a ResNet classifier with and without added unsupervised information from a variational autoencoder (VAE). Although the results do not show a significant improvement, we investigate the effects on the acquisition function and suggest potential approaches for extending the work.

Index Terms—active learning, variational autoencoder, deep learning, pytorch, semi-supervised learning, unsupervised learning

Introduction

In deep learning, the capacity for data gathering often significantly outpaces the labeling. This is easily observed in the field of bioimaging, where ground-truth labeling usually requires the expertise of a clinician. For example, producing a large quantity of CT scans is relatively simple, but having them labeled for COVID-19 by cardiologists takes much more time and money. These constraints ultimately limit the contribution of deep learning to many crucial research problems.

This labeling issue has compelled advancements in the field of active learning (AL) ([Set09]). In a typical AL setting, there is a set of labeled data and a (usually larger) set of unlabeled data. A model is trained on the labeled data, then the model is analyzed to evaluate which unlabeled points should be labeled to best improve the loss objective after further training. AL acknowledges labeling constraints by specifying a budget of points that can be labeled at a time and evaluating against this budget.

In AL, the model for which we select new labels is referred to as the task model. If this model is a classifier neural network, the space in which it maps inputs before classifying them is known as the latent space or representation space. A recent branch of AL ([SS18], [SCN+18], [YK19]), prominent for its applications to deep models, focuses on mapping unlabeled points into the task model's latent space before comparing them.

These methods are limited in their analysis by the labeled data they must train on, failing to make use of potentially useful information embedded in the unlabeled data. We therefore suggest that this family of methods may be improved by extending their representation spaces to include unsupervised features learned over the entire dataset. For this purpose, we opt to use a variational autoencoder (VAE) ([KW13]) , which is a prominent method for unsupervised representation learning. Our main contributions are (a) a new methodology for extending AL methods using VAE features and (b) an experiment comparing AL performance across two recent feature-based AL methods using the new method.

Related Literature

Active learning

Much of the early active learning (AL) literature is based on shallower, less computationally demanding networks since deeper architectures were not well-developed at the time. Settles ([Set09]) provides a review of these early methods. The modern approach uses an acquisition function, which involves ranking all available unlabeled points by some chosen heuristic $\mathscr{H}$ and choosing to label the points of highest ranking.

Algorithm 1: Measuring the performance of a given active learning heuristic

Input: training dataset D, task model $\mathscr{T}$, budget β, initial number of labels ξ, desired number of labels η, set selection heuristic $\mathscr{H}$

$L \leftarrow \xi$-sized random subset of D
$U \leftarrow D \backslash L$
$A \leftarrow \varnothing$
train $\mathscr{T}$ on L
while $|L| \leq \eta$ **do**
  $S \leftarrow \beta$-sized subset of U, selected using $\mathscr{H}$
  $L \leftarrow L \cup S$
  retrain or fine-tune $\mathscr{T}$ on L
  $a \leftarrow$ (validation accuracy of $\mathscr{T}, |L|$) `// save accuracy tuple`
  $A \leftarrow A \cup a$ `// record accuracies across the labeling process`
create a line graph plotting a_0 against a_1 for each $a \in A$

The popularity of the acquisition approach has led to a widely-used evaluation procedure, which we describe in Algorithm 1.

† These authors contributed equally.
* Corresponding author: cmgodwin263@gmail.com, meekail.zain@uga.edu
‡ Institute for Artificial Intelligence, University of Georgia, Athens, GA 30602 USA
* Corresponding author: cmgodwin263@gmail.com, meekail.zain@uga.edu
§ Department of Computer Science, University of Georgia, Athens, GA 30602 USA
¶ Department of Cellular Biology, University of Georgia, Athens, GA 30602 USA

This procedure trains a task model $\mathscr{T}$ on the initial labeled data, records its test accuracy, then uses $\mathscr{H}$ to label a set of unlabeled points. We then once again train $\mathscr{T}$ on the labeled data and record its accuracy. This is repeated until a desired number of labels is reached, and then the accuracies can be graphed against the number of available labels to demonstrate performance over the course of labeling. We can use this evaluation algorithm to separately evaluate multiple acquisition functions on their resulting accuracy graphs. This is utilized in many AL papers to show the efficacy of their suggested heuristics in comparison to others ([WZL+16], [SS18], [SCN+18], [YK19]).

The prevailing approach to point selection has been to choose unlabeled points for which the model is most uncertain, the assumption being that uncertain points will be the most informative ([BRK21]). A popular early method was to label the unlabeled points of highest Shannon entropy ([Sha48]) under the task model, which is a measure of uncertainty between the classes of the data. This method is now more commonly used in combination with a representativeness measure ([WZL+16]) to avoid selecting condensed clusters of very similar points.

Recent heuristics using deep features

For convolutional neural networks (CNNs) in image classification settings, the task model $\mathscr{T}$ can be decomposed into a feature-generating module

$$\mathscr{T}_f \colon \mathbb{R}^n \to \mathbb{R}^f,$$

which maps the input data vectors to the output of the final fully connected layer before classification, and a classification module

$$\mathscr{T}_c \colon \mathbb{R}^f \to \{0, 1, ..., c\},$$

where c is the number of classes.

Recent deep learning-based AL methods have approached the notion of model uncertainty in terms of the rich features generated by the learned model. Core-set ([SS18]) and MedAL ([SCN+18]) select unlabeled points that are the furthest from the labeled set in terms of L_2 distance between the learned features. For core-set, each point constructing the set S in step 6 of Algorithm 1 is chosen by

$$\mathbf{u}^* = \underset{\mathbf{u} \in U}{\operatorname{argmax}} \min_{\boldsymbol{\ell} \in L} ||(\mathscr{T}_f(\mathbf{u}) - \mathscr{T}_f(\boldsymbol{\ell}))||^2, \tag{1}$$

where U is the unlabeled set and L is the labeled set. The analogous operation for MedAL is

$$\mathbf{u}^* = \underset{\mathbf{u} \in U}{\operatorname{argmax}} \frac{1}{|L|} \sum_{i=1}^{|L|} ||\mathscr{T}_f(\mathbf{u}) - \mathscr{T}_f(\mathbf{L_i})||^2. \tag{2}$$

Note that after a point $\mathbf{u}^*$ is chosen, the selection of the next point assumes the previous $\mathbf{u}^*$ to be in the labeled set. This way we discourage choosing sets that are closely packed together, leading to sets that are more diverse in terms of their features. This effect is more pronounced in the core-set method since it takes the minimum distance whereas MedAL uses the average distance.

Another recent method ([YK19]) trains a regression network to predict the loss of the task model, then takes the heuristic $\mathscr{H}$ in Algorithm 1 to select the unlabeled points of highest predicted loss. To implement this, the loss prediction network $\mathscr{P}$ is attached to a ResNet task model $\mathscr{T}$ and is trained jointly with $\mathscr{T}$. The inputs to $\mathscr{P}$ are the features output by the ResNet's four residual blocks. These features are mapped into the same dimensionality via a fully connected layer and then concatenated to form a representation $\mathbf{c}$. An additional fully connected layer then maps $\mathbf{c}$ into a single value constituting the loss prediction.

When attempting to train a network to directly predict $\mathscr{T}$'s loss during training, the ground truth losses naturally decrease as $\mathscr{T}$ is optimized, resulting in a moving objective. The authors of ([YK19]) find that a more stable ground truth is the inequality between the losses of given pairs of points. In this case, $\mathscr{P}$ is trained on pairs of labeled points, so that $\mathscr{P}$ is penalized for producing predicted loss pairs that exhibit a different inequality than the corresponding true loss pair.

More specifically, for each batch of labeled data $L_{batch} \subset L$ that is propagated through $\mathscr{T}$ during training, the batch of true losses is computed and split randomly into a batch of pairs P_{batch}. The loss prediction network produces a corresponding batch of predicted loss pairs, denoted $\widetilde{P}_{batch}$. The following pair loss is then computed given each $p \in P_{batch}$ and its corresponding $\tilde{p} \in \widetilde{P}_{batch}$:

$$\mathscr{L}_{pair}(p, \tilde{p}) = \max(0, -\mathscr{I}(p) \cdot (\tilde{p}^{(1)} - \tilde{p}^{(2)}) + \xi), \tag{3}$$

where $\mathscr{I}$ is the following indicator function for pair inequality:

$$\mathscr{I}(p) = \begin{cases} 1, & p^{(1)} > p^{(2)} \\ -1, & p^{(1)} \leq p^{(2)} \end{cases}. \tag{4}$$

Variational Autoencoders

Variational autoencoders (VAEs) ([KW13]) are an unsupervised method for modeling data using Bayesian posterior inference. We begin with the Bayesian assumption that the data is well-modeled by some distribution, often a multivariate Gaussian. We also assume that this data distribution can be inferred reasonably well by a lower dimensional random variable, also often modeled by a multivariate Gaussian.

The inference process then consists of an encoding into the lower dimensional latent variable, followed by a decoding back into the data dimension. We parametrize both the encoder and the decoder as neural networks, jointly optimizing their parameters with the following loss function ([KW19]):

$$\mathscr{L}_{\theta,\phi}(\mathbf{x}) = \log p_\theta(\mathbf{x}|\mathbf{z}) + [\log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})], \tag{5}$$

where θ and ϕ are the parameters of the encoder and the decoder, respectively. The first term is the reconstruction error, penalizing the parameters for producing poor reconstructions of the input data. The second term is the regularization error, encouraging the encoding to resemble a pre-selected prior distribution, commonly a unit Gaussian prior.

The encoder of a well-optimized VAE can be used to generate latent encodings with rich features which are sufficient to approximately reconstruct the data. The features also have some geometric consistency, in the sense that the encoder is encouraged to generate encodings in the pattern of a Gaussian distribution.

Methods

We observe that the notions of uncertainty developed in the core-set and MedAL methods rely on distances between feature vectors modeled by the task model $\mathscr{T}$. Additionally, loss prediction relies on a fully connected layer mapping from a feature space to a single value, producing different predictions depending on the values of the relevant feature vector. Thus all of these methods utilize spatial reasoning in a vector space.

Furthermore, in each of these methods, the heuristic $\mathscr{H}$ only has access to information learned by the task model, which is

trained only on the labeled points at a given timestep in the labeling procedure. Since variational autoencoder (VAE) encodings are not limited by the contents of the labeled set, we suggest that the aforementioned methods may benefit by expanding the vector spaces they investigate to include VAE features learned across the entire dataset, including the unlabeled data. These additional features will constitute representative and previously inaccessible information regarding the data, which may improve the active learning process.

We implement this by first training a VAE model $\mathscr{V}$ on the given dataset. $\mathscr{V}$ can then be used as a function returning the VAE features for any given datapoint. We append these additional features to the relevant vector spaces using vector concatenation, an operation we denote with the symbol $\frown$. The modified point selection operation in core-set then becomes

$$\mathbf{u}^* = \underset{\mathbf{u}\in U}{\operatorname{argmax}} \min_{\boldsymbol{\ell}\in L} ||([\mathscr{T}_f(\mathbf{u}) \frown \alpha\mathscr{V}(\mathbf{u})] - [\mathscr{T}_f(\boldsymbol{\ell}) \frown \alpha\mathscr{V}(\boldsymbol{\ell})]||^2, \tag{6}$$

where α is a hyperparameter that scales the influence of the VAE features in computing the vector distance. To similarly modify the loss prediction method, we concatenate the VAE features to the final ResNet feature concatenation $\mathbf{c}$ before the loss prediction, so that the extra information is factored into the training of the prediction network $\mathscr{P}$.

Experiments

In order to measure the efficacy of the newly proposed methods, we generate accuracy graphs using Algorithm 1, freezing all settings except the selection heuristic $\mathscr{H}$. We then compare the performance of the core-set and loss prediction heuristics with their VAE-augmented counterparts.

We use ResNet-18 pretrained on ImageNet as the task model, using the SGD optimizer with learning rate 0.001 and momentum 0.9. We train on the MNIST ([Den12]) and ChestMNIST ([YSN21]) datasets. ChestMNIST consists of 112,120 chest X-ray images resized to 28x28 and is one of several benchmark medical image datasets introduced in ([YSN21]).

For both datasets we experiment on randomly selected subsets, using 25000 points for MNIST and 30000 points for ChestMNIST. In both cases we begin with 3000 initial labels and label 3000 points per active learning step. We opt to retrain the task model after each labeling step instead of fine-tuning.

We use a similar training strategy as in ([SCN+18]), training the task model until >99% train accuracy before selecting new points to label. This ensures that the ResNet is similarly well fit to the labeled data at each labeling iteration. This is implemented by training for 10 epochs on the initial training set and increasing the training epochs by 5 after each labeling iteration.

The VAEs used for the experiments are trained for 20 epochs using an Adam optimizer with learning rate 0.001 and weight decay 0.005. The VAE encoder architecture consists of four convolutional downsampling filters and two linear layers to learn the low dimensional mean and log variance. The decoder consists of an upsampling convolution and four size-preserving convolutions to learn the reconstruction.

Experiments were run five times, each with a separate set of randomly chosen initial labels, with the displayed results showing the average validation accuracies across all runs. Figures 1 and 3 show the core-set results, while Figures 2 and 4 show the loss prediction results. In all cases, shared random seeds were used to ensure that the task models being compared were supplied with the same initial set of labels.

With four NVIDIA 2080 GPUs, the total runtime for the MNIST experiments was 5113s for core-set and 4955s for loss prediction; for ChestMNIST, the total runtime was 7085s for core-set and 7209s for loss prediction.

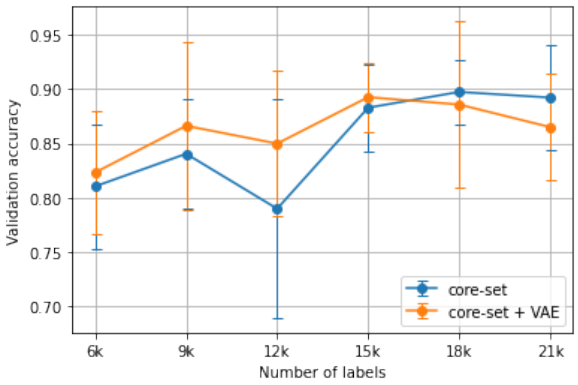


Fig. 1: *The average MNIST results using the core-set heuristic versus the VAE-augmented core-set heuristic for Algorithm 1 over 5 runs.*

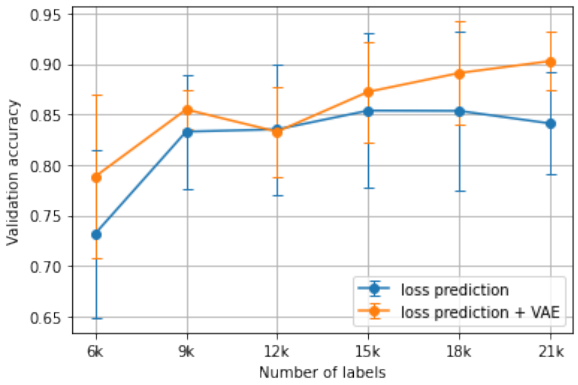


Fig. 2: *The average MNIST results using the loss prediction heuristic versus the VAE-augmented loss prediction heuristic for Algorithm 1 over 5 runs.*

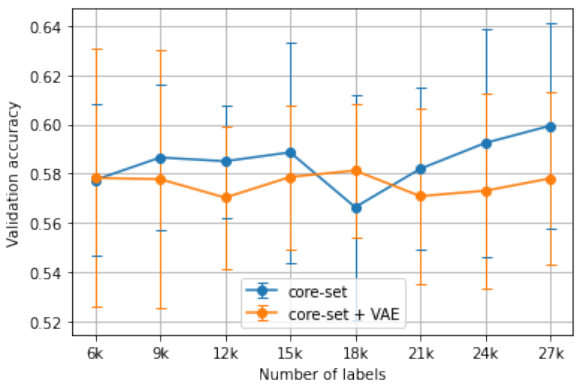


Fig. 3: *The average ChestMNIST results using the core-set heuristic versus the VAE-augmented core-set heuristic for Algorithm 1 over 5 runs.*

To investigate the qualitative difference between the VAE and non-VAE approaches, we performed an additional experiment

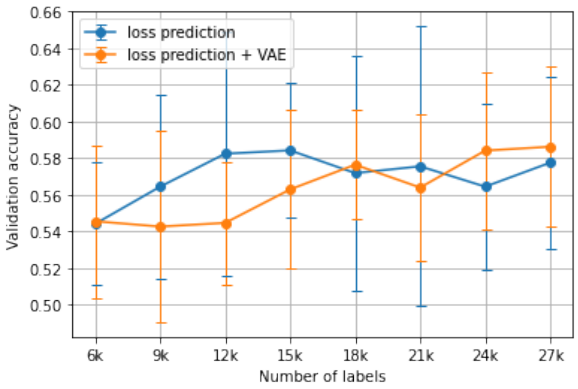


Fig. 4: *The average ChestMNIST results using the loss prediction heuristic versus the VAE-augmented loss prediction heuristic for Algorithm 1 over 5 runs.*

to visualize an example of core-set selection. We first train the ResNet-18 with the same hyperparameter settings on 1000 initial labels from the ChestMNIST dataset, then randomly choose 1556 (5%) of the unlabeled points from which to select 100 points to label. These smaller sizes were chosen to promote visual clarity in the output graphs.

We use t-SNE ([VdMH08]) dimensionality reduction to show the ResNet features of the labeled set, the unlabeled set, and the points chosen to be labeled by core-set.

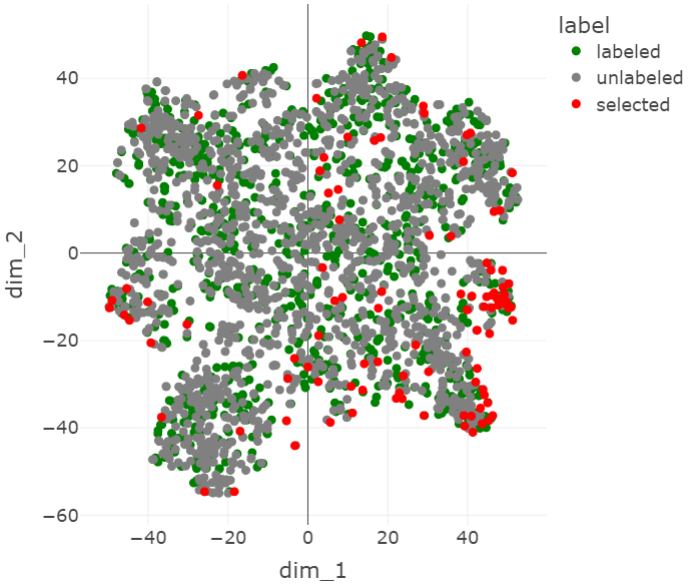


Fig. 5: *A t-SNE visualization of the ChestMNIST points chosen by core-set.*

Discussion

Overall, the VAE-augmented active learning heuristics did not exhibit a significant performance difference when compared with their counterparts. The only case of a significant p-value (<0.05) occurred during loss prediction on the MNIST dataset at 21000 labels.

The t-SNE visualizations in Figures 5 and 6 show some of the influence that the VAE features have on the core-set selection process. In 5, the selected points tend to be more spread out, while in 6 they cluster at one edge. This appears to mirror the transformation of the rest of the data, which is more spread out without the VAE features, but becomes condensed in the center when they are introduced, approaching the shape of a Gaussian distribution.

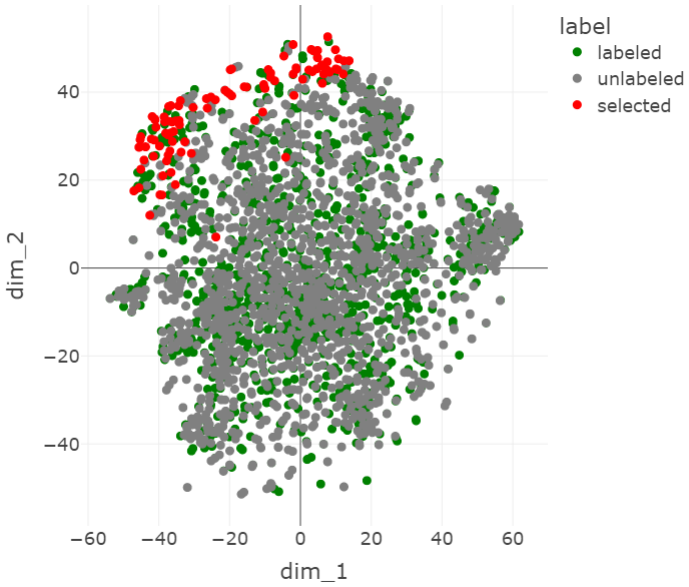


Fig. 6: *A t-SNE visualization of the ChestMNIST points chosen by core-set when the ResNet features are augmented with VAE features.*

It seems that with the added VAE features, the selected points are further out of distribution in the latent space. This makes sense because points tend to be more sparse at the tails of a Guassian distribution and core-set prioritizes points that are well-isolated from other points.

One reason for the lack of performance improvement may be the homogeneous nature of the VAE, where the optimization goal is reconstruction rather than classification. This could be improved by using a multimodal prior in the VAE, which may do a better job of modeling relevant differences between points.

Conclusion

Our original intuition was that additional unsupervised information may improve established active learning methods, especially when using a modern unsupervised representation method such as a VAE. The experimental results did not indicate this hypothesis, but additional investigation of the VAE features showed a notable change in the task model latent space. Though this did not result in superior point selections in our case, it is of interest whether different approaches to latent space augmentation in active learning may fare better.

Future work may explore the use of class-conditional VAEs in a similar application, since a VAE that can utilize the available class labels may produce more effective representations, and it could be retrained along with the task model after each labeling iteration.

References

[BRK21] Samuel Budd, Emma C Robinson, and Bernhard Kainz. A survey on active learning and human-in-the-loop deep learning

for medical image analysis. *Medical Image Analysis*, 71:102062, 2021. doi:10.1016/j.media.2021.102062.

[Den12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. doi:10.1109/MSP.2012.2211477.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[KW19] Diederik P. Kingma and Max Welling. *An Introduction to Variational Autoencoders*. Now Publishers, 2019. URL: https://doi.org/10.1561%2F9781680836233, doi:10.1561/9781680836233.

[SCN+18] Asim Smailagic, Pedro Costa, Hae Young Noh, Devesh Walawalkar, Kartik Khandelwal, Adrian Galdran, Mostafa Mirshekari, Jonathon Fagert, Susu Xu, Pei Zhang, et al. Medal: Accurate and robust deep active learning for medical image analysis. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 481–488. IEEE, 2018. doi:10.1109/icmla.2018.00078.

[Set09] Burr Settles. Active learning literature survey. 2009.

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[SS18] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=H1aIuk-RW.

[VdMH08] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[WZL+16] Keze Wang, Dongyu Zhang, Ya Li, Ruimao Zhang, and Liang Lin. Cost-effective active learning for deep image classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(12):2591–2600, 2016. doi:10.1109/tcsvt.2016.2589879.

[YK19] Donggeun Yoo and In So Kweon. Learning loss for active learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 93–102, 2019. doi:10.1109/CVPR.2019.00018.

[YSN21] Jiancheng Yang, Rui Shi, and Bingbing Ni. Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 191–195, 2021. doi:10.1109/ISBI48211.2021.9434062.

Awkward Packaging: building Scikit-HEP

Henry Schreiner[‡*], Jim Pivarski[‡], Eduardo Rodrigues[§]

Abstract—Scikit-HEP has grown rapidly over the last few years, not just to serve the needs of the High Energy Physics (HEP) community, but in many ways, the Python ecosystem at large. AwkwardArray, boost-histogram/hist, and iminuit are examples of libraries that are used beyond the original HEP focus. In this paper we will look at key packages in the ecosystem, and how the collection of 30+ packages was developed and maintained. Also we will look at some of the software ecosystem contributions made to packages like cibuildwheel, pybind11, nox, scikit-build, build, and pipx that support this effort. We will also discuss the Scikit-HEP developer pages and initial WebAssembly support.

Index Terms—packaging, ecosystem, high energy physics, community project

Introduction

High Energy Physics (HEP) has always had intense computing needs due to the size and scale of the data collected. The World Wide Web was invented at the CERN Physics laboratory in Switzerland in 1989 when scientists in the EU were trying to communicate results and datasets with scientist in the US, and vice-versa [LCC+09]. Today, HEP has the largest scientific machine in the world, at CERN: the Large Hadron Collider (LHC), 27 km in circumference [EB08], with multiple experiments with thousands of collaborators processing over a petabyte of raw data every day, with 100 petabytes being stored per year at CERN. This is one of the largest scientific datasets in the world of exabyte scale [PJ11], which is roughly comparable in order of magnitude to all of astronomy or YouTube [SLF+15].

In the mid nineties, HEP users were beginning to look for a new language to replace Fortran. A few HEP scientists started investigating the use of Python around the release of 1.0.0 in 1994 [Tem22]. A year later, the ROOT project for an analysis toolkit (and framework) was released, quickly making C++ the main language for HEP. The ROOT project also needed an interpreted language to driving analysis code. Python was rejected for this role due to being "exotic" at the time, and because it was considered too much to ask physicists to code in two languages. Instead, ROOT provided a C++ interpreter, called CINT, which later was replaced with Cling, which is the basis for the clang-repl project in LLVM today [IVL22].

Python would start showing up in the late 90's in experiment frameworks as a configuration language. These frameworks were primarily written in C++, but were made of many configurable parts [Lam98]. The glueing together of the system was done in Python, a model still popular today, though some experiments are now using Python + Numba as an alternative model, such as for example the Xenon1T experiment [RTA+17], [RS21].

In the early 2000s, the use of Python HEP exploded, heavily driven by experiments like LHCb developing frameworks and user tools for scripting. ROOT started providing Python bindings in 2004 [LGMM05] that were not considered Pythonic [GTW20], and still required a complex multi-hour build of ROOT to use[1]. Analyses still consisted largely of ROOT, with Python sometimes showing up.

By the mid 2010's, a marked change had occurred, driven by the success of Python in Data Science, especially in education. Many new students were coming into HEP with little or no C++ experience, but with existing knowledge of Python and the growing Python data science ecosystem, like NumPy and Pandas. Several HEP experiment analyses were performed in, or driven by, Python, with ROOT only being used for things that were not available in the Python ecosystem. Some of these were HEP specific: ROOT is also a data format, so users needed to be able to read data from ROOT files. Others were less specific: HEP users have intense histogram requirements due to the data sizes, large portions of HEP data are "jagged" rather than rectangular; vector manipulation was important (especially Lorenz Vectors, a four dimensional relativistic vector with a non-Euclidean metric); and data fitting was important, especially with complex models and accurate error estimation.

Beginnings of a scikit

In 2016, the ecosystem for Python in HEP was rather fragmented. Physicists were developing tools in isolation, without knowing out the overlaps with other tools, and without making them interoperable. There were a handful of popular packages that were useful in HEP spread around among different authors. The ROOTPy project had several packages that made the ROOT-Python bridge a little easier than the built-in PyROOT, such as the root-numpy and related root-pandas packages. The C++ MINUIT fitting library was integrated into ROOT, but the iminuit package [Dea20] provided an easy to install standalone Python package with an extracted copy of MINUIT. Several other specialized standalone C++ packages had bindings as well. Many of the initial authors were transitioning to a less-code centric role or leaving for industry, leaving projects like ROOTPy and iminuit without maintainers.

* Corresponding author: henryfs@princeton.edu
‡ Princeton University
§ University of Liverpool

1. Almost 20 years later ROOT's Python bindings have been rewritten for easier Pythonizations, and installing ROOT in Conda is now much easier, thanks in large part to efforts from Scikit-HEP developers.

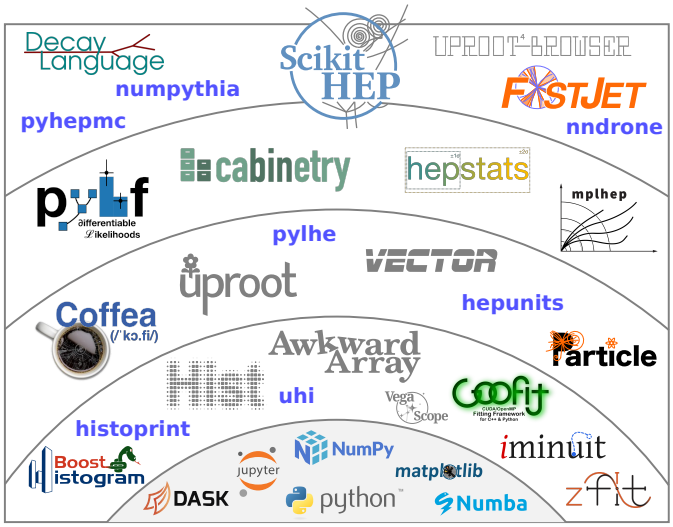


Fig. 1: *The Scikit-HEP ecosystem and affiliated packages.*

Eduardo Rodrigues, a scientist working on the LHCb experiment for the University of Cincinnati, started working on a package called scikit-hep that would provide a set to tools useful for physicists working on HEP analysis. The initial version of the scikit-hep package had a simple vector library, HEP related units and conversions, several useful statistical tools, and provenance recording functionality,

He also placed the scikit-hep GitHub repository in a Scikit-HEP GitHub organization, and asked several of the other HEP related packages to join. The ROOTPy project was ending, with the primary author moving on, and so several of the then-popular packages[2] that were included in the ROOTPy organization were happily transferred to Scikit-HEP. Several other existing HEP libraries, primarily interfacing to existing C++ simulation and tracking frameworks, also joined, like PyJet and NumPythia. Some of these libraries have been retired or replaced today, but were an important part of Scikit-HEP's initial growth.

First initial success

In 2016, the largest barrier to using Python in HEP in a Pythonic way was ROOT. It was challenging to compile, had many non-Python dependencies, was huge compared to most Python libraries, and didn't play well with Python packaging. It was not Pythonic, meaning it had very little support for Python protocols like iteration, buffers, keyword arguments, tab completion and inspect in, dunder methods, didn't follow conventions for useful reprs, and Python naming conventions; it was simply a direct on-demand C++ binding, including pointers. Many Python analyses started with a "convert data" step using PyROOT to read ROOT files and convert them to a Python friendly format like HDF5. Then the bulk of the analysis would use reproducible Python virtual environments or Conda environments.

This changed when Jim Pivarski introduced the Uproot package, a pure-Python implementation of a ROOT file reader (and later writer) that could remove the initial conversion environment by simply pip installing a package. It also had a simple, Pythonic interface and produced outputs Python users could immediately use, like NumPy arrays, instead of PyROOT's wrapped C++ pointers.

Uproot needed to do more than just be file format reader/writer; it needed to provide a way to represent the special structure and common objects that ROOT files could contain. This lead to the development of two related packages that would support uproot. One, uproot-methods, included Pythonic access to functionality provided by ROOT for its core classes, like spatial and Lorentz vectors. The other was AwkwardArray, which would grow to become one of the most important and most general packages in Scikit-HEP. This package allows NumPy-like idioms for array-at-a-time manipulation on jagged data structures. A jagged array is a (possibly structured) array with a variable length dimension. These are very common and relevant in HEP; events have a variable number of tracks, tracks have a variable number of hits in the detector, etc. Many other fields also have jagged data structures. While there are formats to store such structures, computations on jagged structures have usually been closer to SQL queries on multiple tables than direct object manipulation. Pandas handles this through multiple indexing and a lot of duplication.

Uproot was a huge hit with incoming HEP students (see Fig 2); suddenly they could access HEP data using a library installed with pip or conda and no external compiler or library requirements, and could easily use tools they already knew that were compatible with the Python buffer protocol, like NumPy, Pandas and the rapidly growing machine learning frameworks. There were still some gaps and pain points in the ecosystem, but an analysis without writing C++ (interpreted or compiled) and compiling ROOT manually was finally possible. Scikit-HEP did not and does not intend to replace ROOT, but it provides alternative solutions that work natively in the Python "Big Data" ecosystem.

Several other useful HEP libraries were also written. Particle was written for accessing the Particle Data Group (PDG) particle data in a simple and Pythonic way. DecayLanguage originally provided tooling for decay definitions, but was quickly expanded to include tools to read and validate "DEC" decay files, an existing text format used to configure simulations in HEP.

Building compiled packages

In 2018, HEP physicist and programmer Hans Dembinski proposed a histogram library to the Boost libraries, the most influential C++ library collection; many additions to the standard library are based on Boost. Boost.Histogram provided a histogram-as-an-object concept from HEP, but was designed around C++14 templating, using composable axes and storage types. It originally had an initial Python binding, written in Boost::Python. Henry Schreiner proposed the creation of a standalone binding to be written with pybind11 in Scikit-HEP. The original bindings were removed, Boost::Histogram was accepted into the Boost libraries, and work began on boost-histogram. IRIS-HEP, a multi-institution project for sustainable HEP software, had just started, which was providing funding for several developers to work on Scikit-HEP project packages such as this one. This project would pioneer standalone C++ library development and deployment for Scikit-HEP.

There were already a variety of attempts at histogram libraries, but none of them filled the requirements of HEP physicists:

2. The primary package of the ROOTPy project, also called ROOTPy, was not transferred, but instead had a final release and then died. It was an inspiration for the new PyROOT bindings, and influenced later Scikit-HEP packages like mplhep. The transferred libraries have since been replaced by integrated ROOT functionality. All these packages required ROOT, which is not on PyPI, so were not suited for a Python-centric ecosystem.

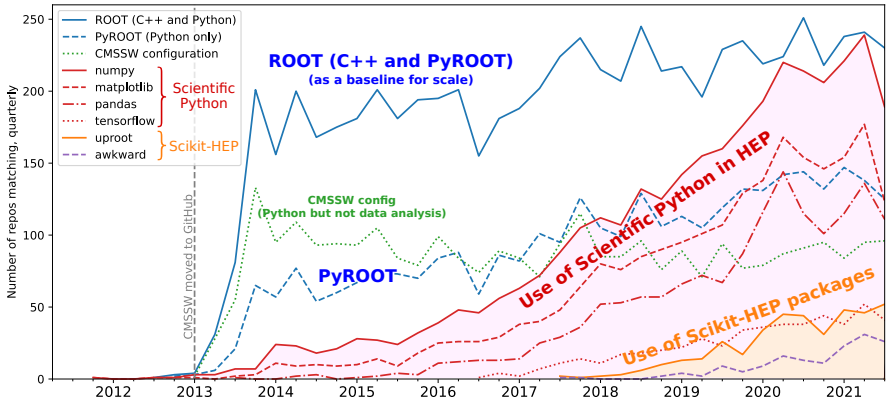


Fig. 2: *Adoption of scientific Python libraries and Scikit-HEP among members of the CMS experiment (one of the four major LHC experiments). CMS requires users to fork* `github:cms-sw/cmssw`*, which can be used to identify 3484 physicist users, who created 16656 non-fork repos. This plot quantifies adoption by counting* `"#include X"`*,* `"import X"`*, and* `"from X import"` *strings in the users' code to measure adoption of various libraries (most popular by category are shown).*

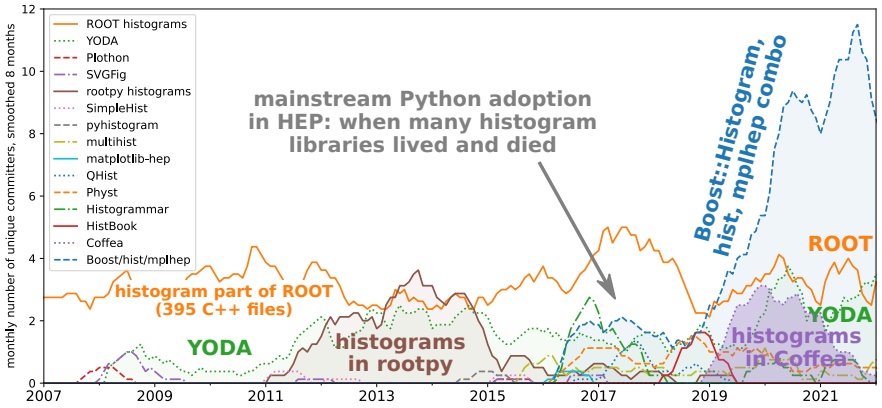


Fig. 3: *Developer activity on histogram libraries in HEP: number of unique committers to each library per month, smoothed (derived from git logs). Illustrates the convergence of a fractured community (around 2017) into a unified one (now).*

fills on pre-existing histograms, simple manipulation of multi-dimensional histograms, competitive performance, and easy to install in clusters or for students. Any new attempt here would have to be clearly better than the existing collection of diverse attempts (see Fig 3). The development of a library with compiled components intended to be usable everywhere required good support for building libraries that was lacking both in Scikit-HEP and to an extent the broader Python ecosystem. Previous advancements in the packaging ecosystem, such as the wheel format for distributing binary platform dependent Python packages and the manylinux specification and docker image that allowed a single compiled wheel to target many distributions of Linux, but there still were many challenges to making a library redistributable on all platforms.

The boost-histogram library only depended on header-only components of the Boost libraries, and the header-only pybind11 package, so it was able to avoid a separate compile step or linking to external dependencies, which simplified the initial build process. All needed files were collected from git submodules and packed into a source distribution (SDist), and everything was built using only setuptools, making build-from-source simple on any system supporting C++14. This did not include RHEL 7, a popular platform in HEP at the time, and on any platform building could take several minutes and required several gigabytes of memory to resolve the heavy C++ templating in the Boost libraries and pybind11.

The first stand-alone development was azure-wheel-helpers, a set of files that helped produce wheels on the new Azure Pipelines platform. Building redistributable wheels requires a variety of techniques, even without shared libraries, that vary dramatically between platforms and were/are poorly documented. On Linux, everything needs to be built inside a controlled manylinux image, and post-processed by the auditwheel tool. On macOS, this includes downloading an official CPython binary for Python to allow older versions of macOS to be targeted (10.9+), several special environment variables, especially when cross compiling to Apple Silicon, and post processing with the delocate tool. Windows is the simplest, as most versions of CPython work identically there. azure-wheel-helpers worked well, and was quickly adapted for the other packages in Scikit-HEP that included non-ROOT binary components. Work here would eventually be merged into the existing and general cibuildwheel package, which would become the build tool for all non-ROOT binary packages in Scikit-HEP, as well as over 600 other packages like matplotlib and numpy, and was accepted into the PyPA (Python Packaging Authority).

The second major development was the upstreaming of CI and build system developments to pybind11. Pybind11 is a C++ API for Python designed for writing a binding to C++, and provided significant benefits to our packages over (mis)-using Cython for bindings; Cython was designed to transpile a Python-

like language to C (or C++), and just happened to support bindings since you can call C and C++ from it, but it was not what it was designed for. Benefits of pybind11 included reduced code complexity and duplication, no pre-process step (cythonize), no need to pin NumPy when building, and a cross-package API. The iMinuit package was later moved from Cython to pybind11 as well, and pybind11 became the Scikit-HEP recommended binding tool. We contributed a variety of fixes and features to pybind11, including positional-only and keyword-only arguments, the option to prepend to the overload chain, and an API for type access and manipulation. We also completely redesigned CMake integration, added a new pure-Setuptools helpers file, and completely redesigned the CI using GitHub Actions, running over 70 jobs on a variety of systems and compilers. We also helped modernize and improve all the example projects with simpler builds, new CI, and cibuildwheel support.

This example of a project with binary components being usable everywhere then encouraged the development of Awkward 1.0, a rewrite of AwkwardArray replacing the Python-only code with compiled code using pybind11, fixing some long-standing limitations, like an inability to slice past two dimensions or select "n choose k" for $k > 5$; these simply could not be expressed using Awkward 0's NumPy expressions, but can be solved with custom compiled kernels. This also enabled further developments in backends [PEL20].

Broader ecosystem

Scikit-HEP had become a "toolset" for HEP analysis in Python, a collection of packages that worked together, instead of a "toolkit" like ROOT, which is one monopackage that tries to provide everything [R+20]. A toolset is more natural in the Python ecosystem, where we have good packaging tools and many existing libraries. Scikit-HEP only needed to fill existing gaps, instead of covering every possible aspect of an analysis like ROOT did. The original scikit-hep package had its functionality pulled out into existing or new separate packages such as HEPUnits and Vector, and the core scikit-hep package instead became a metapackage with no unique functionality on its own. Instead, it installs a useful subset of our libraries for a physicist wanting to quickly get started on a new analysis.

Scikit-HEP was quickly becoming the center of HEP specific Python software (see Fig. 1). Several other projects or packages joined Scikit-HEP iMinuit, a popular HEP and astrophysics fitting library, was probably the most widely used single package to have joined. PyHF and cabinetry also joined; these were larger frameworks that could drive a significant part of an analysis internally using other Scikit-HEP tools.

Other packages, like GooFit, Coffea, and zFit, were not added, but were built on Scikit-HEP packages and had developers working closely with Scikit-HEP maintainers. Scikit-HEP introduced an "affiliated" classification for these packages, which allowed an external package to be listed on the Scikit-HEP website and encouraged collaboration. Coffea had a strong influence on histogram design, and zFit has contributed code to Scikit-HEP. Currently all affiliated packages have at least one Scikit-HEP developer as a maintainer, though that is currently not a requirement. An affiliated package fills a particular need for the community. Scikit-HEP doesn't have to, or need to, attempt to develop a package that others are providing, but rather tries to ensure that the externally provided package works well with the

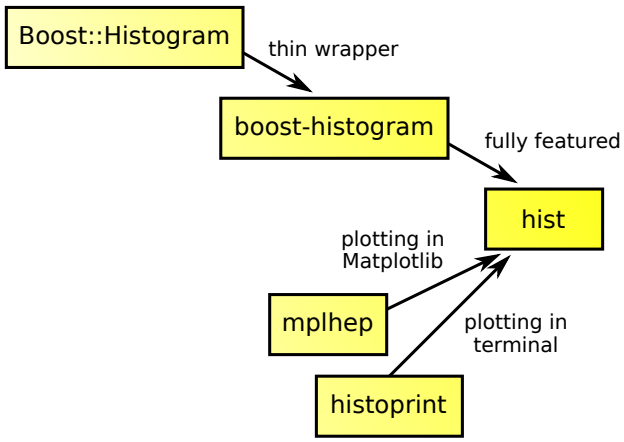


Fig. 4: *The collection of histogram packages and related packages in Scikit-HEP.*

broader HEP ecosystem. The affiliated classification is also used on broader ecosystem packages like pybind11 and cibuildwheel that we recommend and share maintainers with.

Histogramming was designed to be a collection of specialized packages (see Fig. 4) with carefully defined interoperability; boost-histogram for manipulation and filling, Hist for a user-friendly interface and simple plotting tools, histoprint for displaying histograms, and the existing mplhep and uproot packages also needed to be able to work with histograms. This ecosystem was built and is held together with UHI, which is a formal specification agreed upon by several developers of different libraries, backed by a statically typed Protocol, for a PlottableHistogram object. Producers of histograms, like boost-histogram/hist and uproot provide objects that follow this specification, and users of histograms, such as mplhep and histoprint take any object that follows this specification. The UHI library is not required at runtime, though it does also provide a few simple utilities to help a library also accept ROOT histograms, which do not (currently) follow the Protocol, so several libraries have decided to include it at runtime too. By using a static type checker like MyPy to statically enforce a Protocol, libraries that can communicate without depending on each other or on a shared runtime dependency and class inheritance. This has been a great success story for Scikit-HEP, and We expect Protocols to continue to be used in more places in the ecosystem.

The design for Scikit-HEP as a toolset is of many parts that all work well together. One example of a package pulling together many components is uproot-browser, a tool that combines uproot, Hist, and Python libraries like textual and plotext to provide a terminal browser for ROOT files.

Scikit-HEP's external contributions continued to grow. One of the most notable ones was our work on cibuildwheel. This was a Python package that supported building redistributable wheels on multiple CI systems. Unlike our own azure-wheel-helpers or the competing multibuild package, it was written in Python, so good practices in Python package design could apply, like unit and integration tests, static checks, and it was easy to remain independent of the underlying CI system. Building wheels on Linux requires a docker image, macOS requires the python.org Python, and Windows can use any copy of Python - cibuildwheel uses this to supply Python in all cases, which keeps it from

depending on the CI's support for a particular Python version. We merged our improvements to cibuildwheel, like better Windows support, VCS versioning support, and better PEP 518 support. We dropped azure-wheel-helpers, and eventually a scikit-build maintainer joined the cibuildwheel project. cibuildwheel would go on to join the PyPA, and is now in use in over 600 packages, including numpy, matplotlib, mypy, and scikit-learn.

Our continued contributions to cibuildwheel included a TOML-based configuration system for cibuildwheel 2.0, an override system to make supporting multiple manylinux and musllinux targets easier, a way to build directly from SDists, an option to use build instead of pip, the automatic detection of Python version requirements, and better globbing support for build specifiers. We also helped improve the code quality in various ways, including fully statically typing the codebase, applying various checks and style controls, automating CI processes, and improving support for special platforms like CPython 3.8 on macOS Apple Silicon.

We also have helped with build, nox, pyodide, and many other packages, improving the tooling we depend on to develop scikit-build and giving back to the community.

The Scikit-HEP Developer Pages

A variety of packaging best practices were coming out of the boost-histogram work, supporting both ease of installation for users as well as various static checks and styling to keep the package easy to maintain and reduce bugs. These techniques would also be useful apply to Scikit-HEP's nearly thirty other packages, but applying them one-by-one was not scalable. The development and adoption of azure-wheel-helpers included a series of blog posts that covered the Azure Pipelines platform and wheel building details. This ended up serving as the inspiration for a new set of pages on the Scikit-HEP website for developers interested in making Python packages. Unlike blog posts, these would be continuously maintained and extended over the years, serving as a template and guide for updating and adding packages to Scikit-HEP, and educating new developers.

These pages grew to describe the best practices for developing and maintaining a package, covering recommended configuration, style checking, testing, continuous integration setup, task runners, and more. Shortly after the introduction of the developer pages, Scikit-HEP developers started asking for a template to quickly produce new packages following the guidelines. This was eventually produced; the "cookiecutter" based template is kept in sync with the developer pages; any new addition to one is also added to the other. The developer pages are also kept up to date using a CI job that bumps any GitHub Actions or pre-commit versions to the most recent versions weekly. Some portions of the developer pages have been contributed to packaging.python.org, as well.

The cookie cutter was developed to be able to support multiple build backends; the original design was to target both pure Python and Pybind11 based binary builds. This has expanded to include 11 different backends by mid 2022, including Rust extensions, many PEP 621 based backends, and a Scikit-Build based backend for pybind11 in addition to the classic Setuptools one. This has helped work out bugs and influence the design of several PEP 621 packages, including helping with the addition of PEP 621 to Setuptools.

The most recent addition to the pages was based on a new repo-review package which evaluates and existing repository to see what parts of the guidelines are being followed. This was helpful for monitoring adoption of the developer pages, especially newer additions, across the Scikit-HEP packages. This package was then implemented directly into the Scikit-HEP pages, using Pyodide to run Python in WebAssembly directly inside a user's browser. Now anyone visiting the page can enter their repository and branch, and see the adoption report in a couple of seconds.

Working toward the future

Scikit-HEP is looking toward the future in several different areas. We have been working with the Pyodide developers to support WebAssembly; boost-histogram is compiled into Pyodide 0.20, and Pyodide's support for pybind11 packages is significantly better due to that work, including adding support for C++ exception handling. PyHF's documentation includes a live Pyodide kernel, and a try-pyhf site (based on the repo-review tool) lets users run a model without installing anything - it can even be saved as a webapp on mobile devices.

We have also been working with Scikit-Build to try to provide a modern build experience in Python using CMake. This project is just starting, but we expect over the next year or two that the usage of CMake as a first class build tool for binaries in Python will be possible using modern developments and avoiding distutils/setuptools hacks.

Summary

The Scikit-HEP project started in Autumn 2016 and has grown to be a core component in many HEP analyses. It has also provided packages that are growing in usage outside of HEP, like AwkwardArray, boost-histogram/Hist, and iMinuit. The tooling developed and improved by Scikit-HEP has helped Scikit-HEP developers as well as the broader Python community.

REFERENCES

[Dea20] Hans Dembinski and Piti Ongmongkolkul et al. scikit-hep/iminuit. Dec 2020. URL: https://doi.org/10.5281/zenodo.3949207, doi:10.5281/zenodo.3949207.

[EB08] Lyndon Evans and Philip Bryant. Lhc machine. *Journal of instrumentation*, 3(08):S08001, 2008.

[GTW20] Galli, Massimiliano, Tejedor, Enric, and Wunsch, Stefan. "a new pyroot: Modern, interoperable and more pythonic". *EPJ Web Conf.*, 245:06004, 2020. URL: https://doi.org/10.1051/epjconf/202024506004, doi:10.1051/epjconf/202024506004.

[IVL22] Ioana Ifrim, Vassil Vassilev, and David J Lange. GPU Accelerated Automatic Differentiation With Clad. *arXiv preprint arXiv:2203.06139*, 2022.

[Lam98] Stephan Lammel. Computing models of cdf and dØ in run ii. *Computer Physics Communications*, 110(1):32–37, 1998. URL: https://www.sciencedirect.com/science/article/pii/S0010465597001501, doi:10.1016/s0010-4655(97)00150-1.

[LCC+09] Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Larry G Roberts, and Stephen Wolff. A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, 2009.

[LGMM05] W Lavrijsen, J Generowicz, M Marino, and P Mato. Reflection-Based Python-C++ Bindings. 2005. URL: https://cds.cern.ch/record/865620, doi:10.5170/CERN-2005-002.441.

[PEL20] Jim Pivarski, Peter Elmer, and David Lange. Awkward arrays in python, c++, and numba. In *EPJ Web of Conferences*, volume 245, page 05023. EDP Sciences, 2020. doi:10.1051/epjconf/202024505023.

[PJ11] Andreas J Peters and Lukasz Janyst. Exabyte scale storage at CERN. In *Journal of Physics: Conference Series*, volume 331, page 052015. IOP Publishing, 2011. doi:10.1088/1742-6596/331/5/052015.

[R+20] Eduardo Rodrigues et al. The Scikit HEP Project – overview and prospects. *EPJ Web of Conferences*, 245:06028, 2020. arXiv:2007.03577, doi:10.1051/epjconf/202024506028.

[RS21] Olivier Rousselle and Tom Sykora. Fast simulation of Time-of-Flight detectors at the LHC. In *EPJ Web of Conferences*, volume 251, page 03027. EDP Sciences, 2021. doi:10.1051/epjconf/202125103027.

[RTA+17] D Remenska, C Tunnell, J Aalbers, S Verhoeven, J Maassen, and J Templon. Giving pandas ROOT to chew on: experiences with the XENON1T Dark Matter experiment. In *Journal of Physics: Conference Series*, volume 898, page 042003. IOP Publishing, 2017.

[SLF+15] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[Tem22] Jeffrey Templon. Reflections on the uptake of the Python programming language in Nuclear and High-Energy Physics, March 2022. None. URL: https://doi.org/10.5281/zenodo.6353621, doi:10.5281/zenodo.6353621.

Keeping your Jupyter notebook code quality bar high (and production ready) with Ploomber

Ido Michael[‡*]

This paper walks through this interactive tutorial. It is highly recommended running this interactively so it's easier to follow and see the results in real-time. There's a binder link in there as well, so you can launch it instantly.

1. Introduction

Notebooks are an excellent environment for data exploration: they allow us to write code interactively and get visual feedback, providing an unbeatable experience for understanding our data.

However, this convenience comes at a cost; if we are not careful about adding and removing code cells, we may have an irreproducible notebook. Arbitrary execution order is a prevalent problem: a recent analysis found that about 36% of notebooks on GitHub did not execute in linear order. To ensure our notebooks run, we must continuously test them to catch these problems.

A second notable problem is the size of notebooks: the more cells we have, the more difficult it is to debug since there are more variables and code involved.

Software engineers typically break down projects into multiple steps and test continuously to prevent broken and unmaintainable code. However, applying these ideas for data analysis requires extra work; multiple notebooks imply we have to ensure the output from one stage becomes the input for the next one. Furthermore, we can no longer press "Run all cells" in Jupyter to test our analysis from start to finish.

Ploomber provides all the necessary tools to build multi-stage, reproducible pipelines in Jupyter that feel like a single notebook. Users can easily break down their analysis into multiple notebooks and execute them all with a single command.

2. Refactoring a legacy notebook

If you already have a python project in a single notebook, you can use our tool Soorgeon to automatically refactor it into a Ploomber pipeline. Soorgeon statically analyzes your code, cleans up unnecessary imports, and makes sure your monolithic notebook is broken down into smaller components. It does that by scanning the markdown in the notebook and analyzing the headers; each H2 header in our example is marking a new self-contained task.

* Corresponding author: ido@ploomber.io
‡ Ploomber

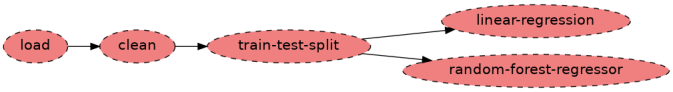


Fig. 1: *In this pipeline none of the tasks were executed - it's all red.*

In addition, it can transform a notebook to a single-task pipeline and then the user can split it into smaller tasks as they see fit.

To refactor the notebook, we use the `soorgeon refactor` command:

```
soorgeon refactor nb.ipynb
```

After running the refactor command, we can take a look at the local directory and see that we now have multiple python tasks which that are ready for production:

```
ls playground
```

We can see that we have a few new files. `pipeline.yaml` contains the pipeline declaration, and `tasks/` contains the *stages* that Soorgeon identified based on our H2 Markdown headings:

```
ls playground/tasks
```

One of the best ways to onboard new people and explain what each workflow is doing is by plotting the pipeline (note that we're now using `ploomber`, which is the framework for developing pipelines):

```
ploomber plot
```

This command will generate the plot below for us, which will allow us to stay up to date with changes that are happening in our pipeline and get the current status of tasks that were executed or failed to execute.

Soorgeon correctly identified the *stages* in our original `nb.ipynb` notebook. It even detected that the last two tasks (`linear-regression`, and `random-forest-regressor`) are independent of each other!

We can also get a summary of the pipeline with `ploomber status`:

```
cd playground
ploomber status
```

3. The `pipeline.yaml` file

To develop a pipeline, users create a `pipeline.yaml` file and declare the tasks and their outputs as follows:

```
Loading pipeline...
name          Last run    Outdated?    Product       Doc (short)   Location
------------  ----------  -----------  ------------  ------------  ------------
load          Has not     Source code  MetaProduct                 /home/jovya
              been run                 ({'df': Fil                 n/playgroun
                                       e('output/l                 d/tasks/loa
                                       oad-                        d.py
                                       df.pkl'),
                                       'nb': File(
                                       'output/loa
                                       d.ipynb')})
clean         Has not     Source code  MetaProduct                 /home/jovya
              been run    & Upstream   ({'df': Fil                 n/playgroun
                                       e('output/c                 d/tasks/cle
                                       lean-                       an.py
                                       df.pkl'),
                                       'nb': File(
                                       'output/cle
                                       an.ipynb')}
                                       )
train-test-   Has not     Source code  MetaProduct                 /home/jovya
split         been run    & Upstream   ({'X_test':                 n/playgroun
                                       File('outpu                 d/tasks/tra
                                       t/...t-X_te                 in-test-
                                       st.pkl'),                   split.py
                                       'X_train':
                                       File('outpu
```

Fig. 2: *In here we can see the status of each of our pipeline's tasks, runtime and location.*

```yaml
tasks:
  - source: script.py
    product:
      nb: output/executed.ipynb
      data: output/data.csv

  # more tasks here...
```

The previous pipeline has a single task (`script.py`) and generates two outputs: `output/executed.ipynb` and `output/data.csv`. You may be wondering why we have a notebook as an output: Ploomber converts scripts to notebooks before execution; hence, our script is considered the source and the notebook a byproduct of the execution. Using scripts as sources (instead of notebooks) makes it simpler to use git. However, this does not mean you have to give up interactive development since Ploomber integrates with Jupyter, allowing you to edit scripts as notebooks.

In this case, since we used `soorgeon` to refactor an existing notebook, we did not have to write the `pipeline.yaml` file.

4. Building the pipeline

Let's build the pipeline (this will take ~30 seconds):

```
cd playground
ploomber build
```

We can see which are the tasks that ran during this command, how long they took to execute, and the contributions of each task to the overall pipeline execution runtime.

Navigate to `playground/output/` and you'll see all the outputs: the executed notebooks, data files and trained model.

```
ls playground/output
```

In this figure, we can see all of the data that was collected during the pipeline, any artifacts that might be useful to the user, and some of the execution history that is saved on the notebook's context.

5. Testing and quality checks

** Open *tasks/train-test-split.py* as a notebook by right-clicking on it and then *Open With -> Notebook* and add the following code after the cell with *# noqa*:

```
[9]: %%sh
ploomber build

Loading pipeline...
name                    Ran?     Elapsed (s)    Percentage
----------------------  ------  -------------  ------------
load                    True          7.04592       20.1485
clean                   True          9.74455       27.8655
train-test-split        True          3.51958       10.0646
linear-regression       True          5.19757       14.863
random-forest-regressor True          9.46236       27.0585
Building task 'load':   0%|          | 0/5 [00:00<?, ?it/s]
Executing:   0%|          | 0/8 [00:00<?, ?cell/s]
Executing:  12%|▉         | 1/8 [00:03<00:25,  3.66s/cell]
Executing:  75%|██████▊   | 6/8 [00:05<00:01,  1.16cell/s]
Executing:  88%|███████▉  | 7/8 [00:06<00:00,  1.38cell/s]
Executing: 100%|█████████| 8/8 [00:06<00:00,  1.14cell/s]
Building task 'clean':  20%|██        | 1/5 [00:07<00:28,  7.05s/it]
Executing:   0%|          | 0/12 [00:00<?, ?cell/s]
Executing:   8%|▊         | 1/12 [00:03<00:41,  3.74s/cell]
Executing:  42%|████      | 5/12 [00:03<00:04,  1.72cell/s]
Executing:  58%|█████▊    | 7/12 [00:04<00:02,  2.20cell/s]
Executing:  75%|███████▌  | 9/12 [00:08<00:03,  1.01s/cell]
Executing: 100%|█████████| 12/12 [00:09<00:00,  1.24cell/s]
Building task 'train-test-split':  40%|████      | 2/5 [00:16<00:25,  8.64s/it]
Executing:   0%|          | 0/9 [00:00<?, ?cell/s]
Executing:  11%|█         | 1/9 [00:02<00:17,  2.16s/cell]
```

Fig. 3: *Here we can see the build outputs*

```
[10]: %%sh
ls output

clean-df.pkl
clean.ipynb
linear-regression.ipynb
load-df.pkl
load.ipynb
random-forest-regressor.ipynb
train-test-split.ipynb
train-test-split-X_test.pkl
train-test-split-X_train.pkl
train-test-split-y_test.pkl
train-test-split-y_train.pkl
```

Fig. 4: *These are the post build artifacts*

```python
# Sample data quality checks after loading the raw data
# Check nulls
assert not df['HouseAge'].isnull().values.any()

# Check a specific range - no outliers
assert df['HouseAge'].between(0,100).any()

# Exact expected row count
assert len(df) == 11085
```

** We'll do the same for *tasks/linear-regression.py*, open the file and add the tests:

```python
# Sample tests after the notebook ran
# Check task test input exists
assert Path(upstream['train-test-split']['X_test']).exists()

# Check task train input exists
assert Path(upstream['train-test-split']['y_train']).exists()

# Validating output type
assert 'pkl' in upstream['train-test-split']['X_test']
```

Adding these snippets will allow us to validate that the data we're looking for exists and has the quality we expect. For instance, in the first test we're checking there are no missing rows, and that the data sample we have are for houses up to 100 years old.

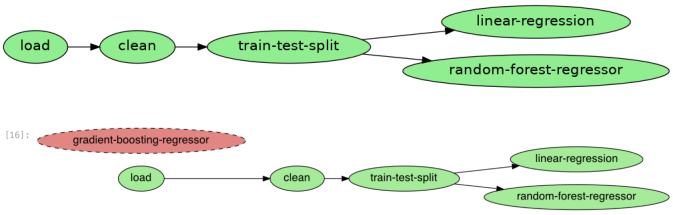


Fig. 5: *Now we see an independent new task*

In the second snippet, we're checking that there are train and test inputs which are crucial for training the model.

6. Maintaining the pipeline

Let's look again at our pipeline plot:

```
Image('playground/pipeline.png')
```

The arrows in the diagram represent input/output dependencies and depict the execution order. For example, the first task (`load`) loads some data, then `clean` uses such data as input and processes it, then `train-test-split` splits our dataset into training and test sets. Finally, we use those datasets to train a linear regression and a random forest regressor.

Soorgeon extracted and declared this dependencies for us, but if we want to modify the existing pipeline, we need to declare such dependencies. Let's see how.

We can also see that the pipeline is green, meaning all of the tasks in it have been executed recently.

7. Adding a new task

Let's say we want to train another model and decide to try Gradient Boosting Regressor. First, we modify the `pipeline.yaml` file and add a new task:

Open `playground/pipeline.yaml` and add the following lines at the end

```
- source: tasks/gradient-boosting-regressor.py
  product:
    nb: output/gradient-boosting-regressor.ipynb
```

Now, let's create a base file by executing `ploomber scaffold`:

```
cd playground
ploomber scaffold
```

This is the output of the command: `` ` Found spec at 'pipeline.yaml' Adding /Users/ido/ploomber-workshop/playground/tasks/ gradient-boosting-regressor.py... Created 1 new task sources. ` ``

We can see it created the task sources for our new task, we just have to fill those in right now.

Let's see how the plot looks now:

```
cd playground
ploomber plot
```

You can see that Ploomber recognizes the new file, but it does not have any dependency, so let's tell Ploomber that it should execute after `train-test-split`:

Open

```
playground/tasks/gradient-boosting-regressor.py
```

as a notebook by right-clicking on it and then `Open With` -> `Notebook`:

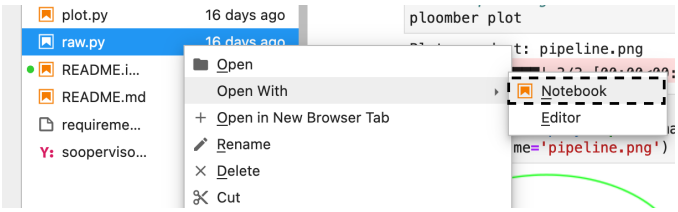


Fig. 6: *lab-open-with-notebook*

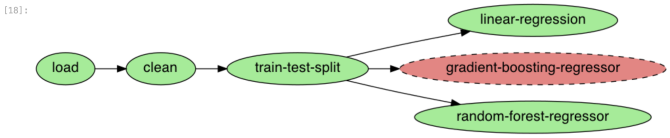


Fig. 7: *The new task is attached to the pipeline*

At the top of the notebook, you'll see the following:

```
upstream = None
```

This special variable indicates which tasks should execute before the notebook we're currently working on. In this case, we want to get training data so we can train our new model so we change the `upstream` variable:

```
upstream = ['train-test-split']
```

Let's generate the plot again:

```
cd playground
ploomber plot
```

Ploomber now recognizes our dependency declaration!

Open

```
playground/tasks/gradient-boosting-regressor.py
```

as a notebook by right-clicking on it and then `Open With` -> `Notebook` and add the following code:

```
from pathlib import Path
import pickle

import seaborn as sns
from sklearn.ensemble import GradientBoostingRegressor

y_train = pickle.loads(Path(
    upstream['train-test-split']['y_train']).read_bytes())
y_test = pickle.loads(Path(
    upstream['train-test-split']['y_test']).read_bytes())
X_test = pickle.loads(Path(
    upstream['train-test-split']['X_test']).read_bytes())
X_train = pickle.loads(Path(
    upstream['train-test-split']['X_train']).read_bytes())

gbr = GradientBoostingRegressor()
gbr.fit(X_train, y_train)

y_pred = gbr.predict(X_test)
sns.scatterplot(x=y_test, y=y_pred)
```

8. Incremental builds

Data workflows require a lot of iteration. For example, you may want to generate a new feature or model. However, it's wasteful to re-execute every task with every minor change. Therefore, one of Ploomber's core features is incremental builds, which automatically skip tasks whose source code hasn't changed.

Run the pipeline again:

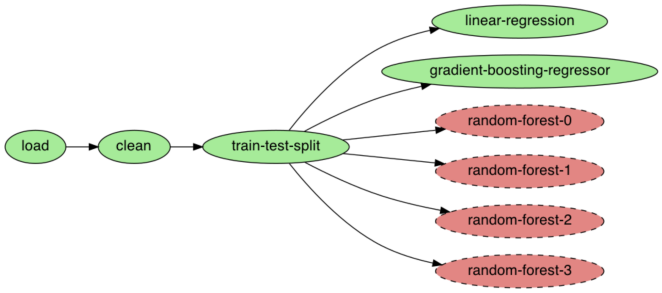


Fig. 8: *We can see this pipeline has multiple new tasks.*

```
cd playground
ploomber build
```

You can see that only the `gradient-boosting-regressor` task ran!

Incremental builds allow us to iterate faster without keeping track of task changes.

Check out `playground/output/gradient-boosting-regressor.ipynb`, which contains the output notebooks with the model evaluation plot.

9. Parallel execution and Ploomber cloud execution

This section can run locally or on the cloud. To setup the cloud we'll need to register for an api key

Ploomber cloud allows you to scale your experiments into the cloud without provisioning machines and without dealing with infrastrucutres.

Open *playground/pipeline.yaml* and add the following code instead of the source task:

```
- source: tasks/random-forest-regressor.py
```

This is how your task should look like in the end

```
- source: tasks/random-forest-regressor.py
  name: random-forest-
  product:
    nb: output/random-forest-regressor.ipynb
  grid:
        # creates 4 tasks (2 * 2)
        n_estimators: [5, 10]
        criterion: [gini, entropy]
```

In addition, we'll need to add a flag to tell the pipeline to execute in parallel. Open *playground/pipeline.yaml* and add the following code above the *-tasks* section (line 1):

yaml

Execute independent tasks in parallel executor: parallel

```
ploomber plot

ploomber build
```

10. Execution in the cloud

When working with datasets that fit in memory, running your pipeline is simple enough, but sometimes you may need more computing power for your analysis. Ploomber makes it simple to execute your code in a distributed environment without code changes.

Check out Soopervisor, the package that implements exporting Ploomber projects in the cloud with support for:

- Kubernetes (Argo Workflows)
- AWS Batch
- Airflow

11. Resources

Thanks for taking the time to go through this tutorial! We hope you consider using Ploomber for your next project. If you have any questions or need help, please reach out to us! (contact info below).

Here are a few resources to dig deeper:

- GitHub
- Documentation
- Code examples
- JupyterCon 2020 talk
- Argo Community Meeting talk
- Pangeo Showcase talk (AWS Batch demo)
- Jupyter project

10. Contact

- Twitter
- Join us on Slack
- E-mail us

Likeness: a toolkit for connecting the social fabric of place to human dynamics

Joseph V. Tuccillo[‡*], James D. Gaboardi[‡]

Abstract—The ability to produce richly-attributed synthetic populations is key for understanding human dynamics, responding to emergencies, and preparing for future events, all while protecting individual privacy. The Likeness toolkit accomplishes these goals with a suite of Python packages: `pymedm`/`pymedm_legacy`, `livelike`, and `actlike`. This production process is initialized in `pymedm` (or `pymedm_legacy`) that utilizes census microdata records as the foundation on which disaggregated spatial allocation matrices are built. The next step, performed by `livelike`, is the generation of a fully autonomous agent population attributed with hundreds of demographic census variables. The agent population synthesized in `livelike` is then attributed with residential coordinates in `actlike` based on block assignment and, finally, allocated to an optimal daytime activity location via the street network. We present a case study in Knox County, Tennessee, synthesizing 30 populations of public K–12 school students & teachers and allocating them to schools. Validation of our results shows they are highly promising by replicating reported school enrollment and teacher capacity with a high degree of fidelity.

Index Terms—activity spaces, agent-based modeling, human dynamics, population synthesis

Introduction

Human security fundamentally involves the functional capacity that individuals possess to withstand adverse circumstances, mediated by the social and physical environments in which they live [Hew97]. Attention to human dynamics is a key piece of the human security puzzle, as it reveals spatial policy interventions most appropriate to the ways in which people within a community behave and interact in daily life. For example, "one size fits all" solutions do not exist for mitigating disease spread, promoting physical activity, or enabling access to healthy food sources. Rather, understanding these outcomes requires examination of processes like residential sorting, mobility, and social transmission.

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

Modeling these processes at scale and with respect to individual privacy is most commonly achieved through agent-based simulations on *synthetic populations* [SEM14]. Synthetic populations consist of individual agents that, when viewed in aggregate, closely recreate the makeup of an area's observed population [HHSB12], [TMKD17]. Modeling human dynamics with synthetic populations is common across research areas including spatial epidemiology [DKA+08], [BBE+08], [HNB+11], [NCA13], [RSF+21], [SNGJ+09], public health [BCD+06], [BFH+17], [SPH11], [TCR08], [MCB+08], and transportation [BBM96], [ZFJ14]. However, a persistent limitation across these applications is that synthetic populations often do not capture a wide enough range of individual characteristics to assess how human dynamics are linked to human security problems (e.g., how a person's age, limited transportation access, and linguistic isolation may interact with their housing situation in a flood evacuation emergency).

In this paper, we introduce Likeness [TG22], a Python toolkit for connecting the social fabric of place to human dynamics via models that support increased spatial, temporal, and demographic fidelity. Likeness is an extension of the UrbanPop framework developed at Oak Ridge National Laboratory (ORNL) that embraces a new paradigm of "vivid" synthetic populations [TM21], [Tuc21], in which individual agents may be attributed in potentially hundreds of ways, across subjects spanning demographics, socioeconomic status, housing, and health. Vivid synthetic populations benefit human dynamics research both by enabling more precise geolocation of population segments, as well as providing a deeper understanding of how individual and neighborhood characteristics are coupled. UrbanPop's early development was motivated by linking models of residential sorting and worker commute behaviors [MNP+17], [MPN+17], [ANM+18]. Likeness expands upon the UrbanPop approach by providing a novel integrated model that pairs vivid residential synthetic populations with an activity simulation model on real-world transportation networks, with travel destinations based on points of interest (POIs) curated from location services and federal critical facilities data.

We first provide an overview of Likeness' capabilities, then provide a more detailed walkthrough of its central workflow with respect to `livelike`, a package for population synthesis and residential characterization, and `actlike` a package for activity allocation. We provide preliminary usage examples for Likeness based on 1) social contact networks in POIs 2) 24-hour POI occupancy characteristics. Finally, we discuss existing limitations and the outlook for future development.

Overview of Core Capabilities and Workflow

UrbanPop initially combined the vivid synthetic populations produced from the American Community Survey (ACS) using the *Penalized-Maximum Entropy Dasymetric Modeling* (P-MEDM) method, which is detailed later, with a commute model based on origin-destination flows, to generate a detailed dataset of daytime and nighttime synthetic populations across the United States [MPN+17]. Our development of Likeness is motivated by extending the existing capabilities of UrbanPop to routing libraries available in Python like `osmnx`[1] and `pandana`[2] [Boe17], [FW12]. In doing so, we are able to simulate travel to regular daytime activities (work and school) based on real-world transportation networks. Likeness continues to use the P-MEDM approach, but is fully integrated with the U.S. Census Bureau's ACS Summary File (SF) and Census Microdata APIs, enabling the production of activity models on-the-fly.

Likeness features three core capabilities supporting activity simulation with vivid synthetic populations (Figure 1). The first, spatial allocation, is provided by the `pymedm` and `pmedm_legacy` packages and uses Iterative Proportional Fitting (IPF) to downscale census microdata records to small neighborhood areas, providing a basis for population synthesis. Baseline residential synthetic populations are then created and stratified into agent segments (e.g., grade 10 students, hospitality workers) using the `livelike` package. Finally, the `actlike` package models travel across agent segments of interest to POIs outside places of residence at varying times of day.

Spatial Allocation: the `pymedm` & `pmedm_legacy` packages

Synthetic populations are typically generated from census microdata, which consists of a sample of publicly available longform responses to official statistical surveys. To preserve respondent confidentiality, census microdata is often published at spatial scales the size of a city or larger. Spatial allocation with IPF provides a maximum-likelihood estimator for microdata responses in small (e.g., neighborhood) areas based on aggregate data published about those areas (known as "constraints"), resulting in a baseline for population synthesis [WCC+09], [BBM96], [TMKD17]. UrbanPop is built upon a regularized implementation of IPF, the P-MEDM method, that permits many more input census variables than traditional approaches [LNB13], [NBLS14]. The P-MEDM objective function (Eq. 1) is written as:

$$\max -\sum_{it} \frac{n}{N} \frac{w_{it}}{d_{it}} \log \frac{w_{it}}{d_{it}} - \sum_{k} \frac{e_k^2}{2\sigma_k^2} \tag{1}$$

where w_{it} is the estimate of variable i in zone t, d_{it} is the synthetic estimate of variable i in location t, n is the number of microdata responses, and N is the total population size. Uncertainty in variable estimates is handled by adding an error term to the allocation $\sum_k \frac{e_k^2}{2\sigma_k^2}$, where e_k is the error between the synthetic and published estimate of ACS variable k and σ_k is the ACS standard error for the estimate of variable k. This is accomplished by leveraging the uncertainty in the input variables: the "tighter" the margins of error on the estimate of variable k in place t, the more leverage it holds upon the solution [NBLS14].

The P-MEDM procedure outputs an *allocation matrix* that estimates the probability of individuals matching responses from the ACS Public-Use Microdata Sample (PUMS) at the scale of census block groups (typically 300–6000 people) or tracts (1200–8000 people), depending upon the use-case.

Downscaling the PUMS from the Public-Use Microdata Area (PUMA) level at which it is offered (100,000 or more people) to these neighborhood scales then enables us to produce synthetic populations (the `livelike` package) and simulate their travel to POIs (the `actlike` package) in an integrated model. This approach provides a new means of modeling population mobility and activity spaces with respect to real-world transportation networks and POIs, in turn enabling investigation of social processes from the atomic (e.g., person) level in human systems.

Likeness offers two implementations of P-MEDM. The first, the `pymedm` package, is written natively in Python based on `scipy.optimize.minimize`, and while fully operational remains in development and is currently suitable for one-off simulations. The second, the `pmedm_legacy` package, uses `rpy2` as a bridge to [NBLS14]'s original implementation of P-MEDM[3] in R/C++ and is currently more stable and scalable. We offer `conda` environments specific to each package, based on user preferences.

Each package's functionality centers around a `PMEDM` class, which contains information required to solve the P-MEDM problem:

- The individual (household) level constraints based on ACS PUMS. To preserve households from the PUMS in the synthetic population, the person-level constraints describing household members are aggregated to the household level and merged with household-level constraints.
- PUMS household sample weights.
- The target (e.g., block group) and aggregate (e.g., tract) zone constraints based on population-level estimates available in the ACS SF.
- The target/aggregate zone 90% margins of error and associated standard errors ($SE = 1.645 \times MOE$).

The `PMEDM` classes feature a `solve()` method that returns an optimized P-MEDM solution and allocation matrix. Through a `diagnostics` module, users may then evaluate a P-MEDM solution based on the proportion of published 90% MOEs from the summary-level ACS data preserved at the target (allocation) scale.

Population Synthesis: the `livelike` package

The `livelike` package generates baseline residential synthetic populations and performs agent segmentation for activity simulation.

Specifying and Solving Spatial Allocation Problems

The `livelike` workflow is oriented around a user-specified *constraints* file containing all of the information necessary to specify a P-MEDM problem for a PUMA of interest. "Constraints" are variables from the ACS common among people/households (PUMS) and populations (SF) that are used as both model inputs and descriptors. The constraints file includes information for bridging PUMS variable definitions with those from the SF using helper functions provided by the `livelike.pums` module, including table IDs, sampling universe (person/household), and tags for the range of ACS vintages (years) for which the variables are relevant.

1. https://github.com/gboeing/osmnx
2. https://github.com/UDST/pandana
3. https://bitbucket.org/nnnagle/pmedmrcpp

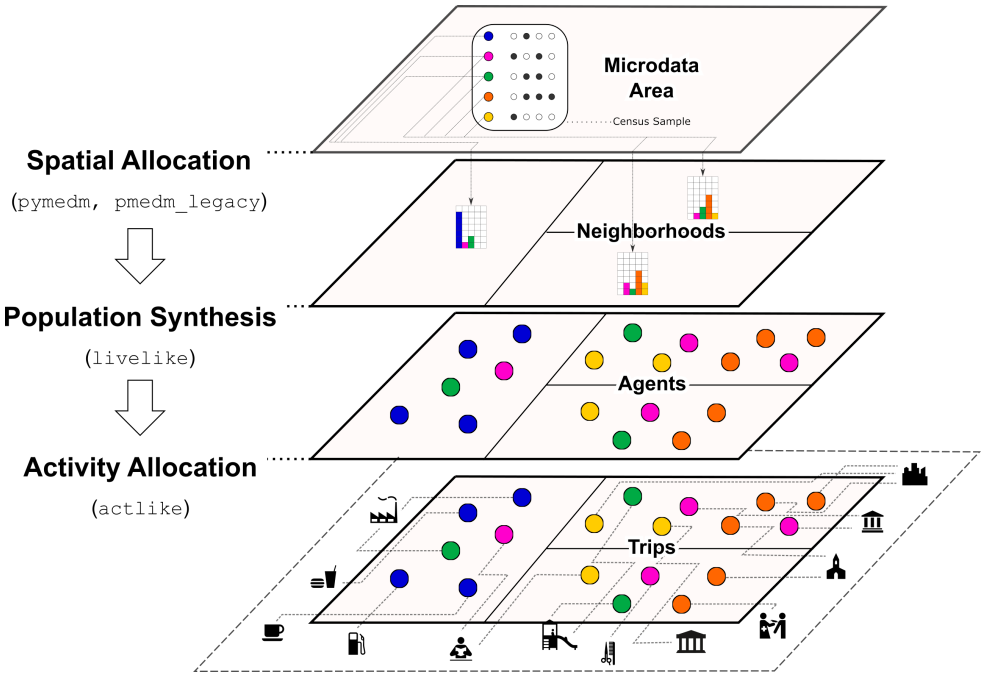


Fig. 1: *Core capabilities and workflow of Likeness.*

The primary `livelike` class is the `acs.puma`, which stores information about a single PUMA necessary for spatial allocation of the PUMS data to block groups/tracts with P-MEDM. The process of creating an `acs.puma` is integrated with the U.S. Census Bureau's ACS SF and Census Microdata 5-Year Estimates (5YE) APIs[4]. This enables generation of an `acs.puma` class with a high-level call involving just a few parameters: 1) the PUMA's Federal Information Processing Standard (FIPS) code 2) the constraints file, loaded as a `pandas.DataFrame` and 3) the target ACS vintage (year). An example call to build an `acs.puma` for the Knoxville City, TN PUMA (FIPS 4701603) using the ACS 2015–2019 5-Year Estimates is:

```
acs.puma(
    fips="4701603",
    constraints=constraints,
    year=2019
)
```

The `censusdata` package[5] is used internally to fetch population-level (SF) constraints, standard errors, and MOEs from the ACS 5YE API, while the `acs.extract_pums_constraints` function is used to fetch individual-level constraints and weights from the Census Microdata 5YE API.

Spatial allocation is then carried out by passing the `acs.puma` attributes to a `pymedm.PMEDM` or `pmedm_legacy.PMEDM` (depending on user preference).

Population Synthesis

The `homesim` module provides support for population synthesis on the spatial allocation matrix within a solved `P-MEDM` object. The population synthesis procedure involves converting the fractional estimates from the allocation matrix (n household IDs by m zones) to integer representation such that whole people/households are preserved. This `homesim` module features an implementation of [LB13]'s "Truncate, Replicate, Sample" (TRS) method. TRS works by separating each cell of the allocation matrix into whole-number (integer) and fractional components, then incrementing the whole-number estimates by a random sample of unit weights performed with sampling probabilities based on the fractional component. Because TRS is stochastic, the `homesim.hsim()` function generates multiple (default 30) realizations of the residential population. The results are provided as a `pandas.DataFrame` in long format, attributed by:

- PUMS Household ID (`h_id`)
- Simulation number (`sim`)
- Target zone FIPS code (`geoid`)
- Household count (`count`)

Since household and person-level attributes are combined when creating the `acs.puma` class, person-level records from the PUMS are assumed to be joined to the synthesized household IDs many-to-one. For example, if two people, $A01$ and $A03$, in household A have some attribute of interest, and there are 3 households of type A in zone G, then we estimate that a total of 6 people with that attribute from household A reside in zone G.

Agent Generation

The synthetic populations can then be segmented into different groups of agents (e.g., workers by industry, students by grade) for activity modeling with the `actlike` package. Agent segments may be identified in several ways:

- Using `acs.extract_pums_segment_ids()` to fetch the person IDs (household serial number + person line number) from the Census Microdata API matching some criteria of interest (e.g., public school students in 10^{th} grade).
- Using `acs.extract_pums_descriptors()` to fetch criteria that may be queried from the Census Microdata API. This is useful when dealing with criteria

4. https://www.census.gov/data/developers/data-sets.html
5. https://pypi.org/project/CensusData

more specific than can be directly controlled for in the P-MEDM problem (e.g., detailed NAICS code of worker, exact number of hours worked).

The function `est.tabulate_by_serial()` is then used to tabulate agents by target zone and simulation by appending them to the synthetic population based on household ID, then aggregating the person-level counts. This routine is flexible in that a user can use any set of criteria available from the PUMS to define customized agents for mobility modeling purposes.

Other Capabilities

Population Statistics: In addition to agent creation, the `livelike.est` module also supports the creation of population statistics. This can be used to estimate the compositional characteristics of small neighborhood areas and POIs, for example to simulate social contact networks (see Students). To accomplish this, the results of `est.tabulate_by_serial` (see Agent Generation) are converted to proportional estimates to facilitate POIs (`est.to_prop()`), then averaged across simulations to produce Monte Carlo estimates and errors `est.monte_carlo_estimate()`).

Multiple ACS Vintages and PUMAs: The `multi` module extends the capabilities of `livelike` to multiple ACS 5YE vintages (dating back to 2016), as well as multiple PUMAs (e.g., a metropolitan area) via the `multi` module. Using `multi.make_pumas()` or `multi.make_multiyear_pumas()`, multiple PUMAs/multiple years may be stored in a `dict` that enables iterative runs for spatial allocation (`multi.make_pmedm_problems()`), population synthesis (`multi.homesim()`), and agent creation (`multi.extract_pums_segment_ids()`, `multi.extract_pums_segment_ids_multiyear()`, `multi.extract_pums_descriptors()`, and `multi.extract_pums_descriptors_multiyear()`). This functionality is currently available for `pmedm_legacy` only.

Activity Allocation: the `actlike` package

The `actlike` package [GT22] allocates agents from synthetic populations generated by `livelike` POI, like schools and workplaces, based on optimal allocation about transportation networks derived from `osmnx` and `pandana` [Boe17], [FW12]. Solutions are the product of a modified integer program (Transportation Problem [Hit41], [Koo49], [MS01], [MS15]) modeled in `pulp` or `mip` [MOD11], [ST20], whereby supply (students/workers) are "shipped" to demand locations (schools/workplaces), with potentially relaxed minimum and maximum capacity constraints at demand locations. Impedance from nighttime to daytime locations (Origin-Destination [OD] pairs) can be modeled by either network distance or network travel time.

Location Synthesis

Following the generation of synthetic households for the study universe, locations for all households across the 30 default simulations must be created. In order to intelligently site pseudo-neighborhood clusters of random points, we adopt a dasymetric [QC13] approach, which we term *intelligent block-based* (IBB) allocation, whereby household locations are only placed within blocks known to have been populated at a particular period in time and are placed with a greater frequency proportional to reported household density [LB13]. We employ population and housing counts within 2010 Decennial Census blocks to formulate a modified Variable Size Bin Packing Problem [FL86], [CGSdG08] for each populated block group, which allows for an optimal placement of household points and is accomplished by the `actlike.block_denisty_allocation()` function that creates and solves an `actlike.block_allocation.BinPack` instance.

Activity Allocation

Once household location attribution is complete, individual agents must be allocated from households (nighttime locations) to probable activity spaces (daytime locations). This is achieved through spatial network modeling over the streets within a study area via OpenStreetMap[6] utilizing `osmnx` for network extraction & preprocessing and `pandana` for shortest path and route calculations. The underlying impedance metric for shortest path calculation, handled in `actlike.calc_cost_mtx()` and associated internal functions, can either take the form of distance or travel time. Moreover, household and activity locations must be connected to nearby network edges for realistic representations within network space [GFH20].

With a cost matrix from all residences to daytime locations calculated, the simulated population can then be "sent" to the likely activity spaces by utilizing an instance of `actlike.ActivityAllocation` to generate an adapted Transportation Problem. This mixed integer program, solved using the `solve()` method, optimally associates all population within an activity space with the objective of minimizing the total cost of impedance (Eq. 2), being subject to potentially relaxed minimum and maximum capacity constraints (Eq. 4 & 5). Each decision variable (x_{ij}) represents a potential allocation from origin i to destination j that must be an integer greater than or equal to zero (Eq. 6 & 7). The problem is formulated as follows:

$$\min \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{2}$$

$$\text{s.t.} \quad \sum_{j \in J} x_{ij} = O_i \quad \forall i \in I; \tag{3}$$

$$\text{s.t.} \quad \sum_{i \in I} x_{ij} \geq minD_j \quad \forall j \in J; \tag{4}$$

$$\text{s.t.} \quad \sum_{i \in I} x_{ij} \leq maxD_j \quad \forall j \in J; \tag{5}$$

$$\text{s.t.} \quad x_{ij} \geq 0 \quad \forall i \in I \quad \forall j \in J; \tag{6}$$

$$\text{s.t.} \quad x_{ij} \in \mathbb{Z} \quad \forall i \in I \quad \forall j \in J. \tag{7}$$

where

$i \in I$ = each household in the set of origins
$j \in J$ = each school in the set of destinations
x_{ij} = allocation decision from $i \in I$ to $j \in J$
c_{ij} = cost between all i, j pairs
O_i = population in origin i for $i \in I$
$minD_j$ = minimum capacity j for $j \in J$
$maxD_j$ = maximum capacity j for $j \in J$

6. https://www.openstreetmap.org/about

The key to this adapted formulation of the classic Transportation Problem is the utilization of minimum and maximum capacity thresholds that are generated endogenously within `actlike.ActivityAllocation` and are tuned to reflect the uncertainty of both the population estimates generated by `livelike` and the reported (or predicted) capacities at activity locations. Moreover, network impedance from origins to destinations (c_{ij}) can be randomly reduced through an internal process by passing in an integer value to the `reduce_seed` keyword argument. By triggering this functionality, the count and magnitude of reduction is determined algorithmically. A random reduction of this nature is beneficial in generating dispersed solutions that do not resemble compact clusters, with an example being the replication of a private school's student body that does not adhere to public school attendance zones.

After the optimal solution is found for an `actlike.ActivityAllocation` instance, selected decisions are isolated from non-zero decision variables with the `realized_allocations()` method. These allocations are then used to generate solution routes with the `network_routes()` function that represent the shortest path along the network traversed from residential locations to assigned activity spaces. Solutions can be further validated with Canonical Correlation Analysis, in instances where the agent segments are stratified, and simple linear regression for those where a single segment of agents is used. Validation is discussed further in Validation & Diagnostics.

Case Study: K–12 Public Schools in Knox County, TN

To illustrate Likeness' capability to simulate POI travel among specific population segments, we provide a case study of travel to POIs, in this case K–12 schools, in Knox County, TN. Our choice of K–12 schools was motivated by several factors. First, they serve as common destinations for the two major groups—workers and students—expected to consistently travel on a typical business day [RWM+17]. Second, a complete inventory of public school locations, as well as faculty and enrollment sizes, is available publicly through federal open data sources. In this case, we obtained school locations and faculty sizes from the Homeland Infrastructure Foundation-Level Database (HIFLD)[7] and student enrollment sizes by grade from the National Center for Education Statistics (NCES) Common Core of Data[8].

We chose the Knox County School District, which coincides with Knox County's boundaries, as our study area. We used the `livelike` package to create 30 synthetic populations for the Knoxville Core-Based Statistical Area (CBSA), then for each simulation we:

- Isolated agent segments from the synthetic population. K–12 educators consist of full-time workers employed as primary and secondary education teachers (2018 Standard Occupation Classification System codes 2300–2320) in elementary and secondary schools (NAICS 6111). We separated out student agents by public schools and by grade level (Kindergarten through Grade 12).
- Performed *IBB* allocation to simulate the household locations of workers and students. Our selection of household locations for workers and students varied geographically. Because school attendance in Knox County is restricted by district boundaries, we only placed student households in the PUMAs intersecting with the district (FIPS 4701601, 4701602, 4701603, 4701604). However, because educators may live outside school district boundaries, we simulated their household locations throughout the Knoxville CBSA.
- Used `actlike` to perform optimal allocation of workers and students about road networks in Knox County/Knoxville CBSA. Across the 30 simulations and 14 segments identified, we produced a total of 420 travel simulations. Network impedance was measured in geographic distance for all student simulations and travel time for all educator simulations.

Figure 2 demonstrates the optimal allocations, routing, and network space for a single simulation of 10th grade public school students in Knox County, TN. Students, shown in households as small black dots, are associated with schools, represented by transparent colored circles sized according to reported enrollment. The network space connecting student residential locations to assigned schools is displayed in a matching color. Further, the inset in Figure 2 provides the pseudo-school attendance zone for 10th graders at one school in central Knoxville and demonstrates the adherence to network space.

Students

Our study of K–12 students examines social contact networks with respect to potentially underserved student populations via the compositional characteristics of POIs (schools).

We characterized each school's student body by identifying student profiles based on several criteria: minority race/ethnicity, poverty status, single caregiver households, and unemployed caregiver households (householder and/or spouse/parnter). We defined 6 student profiles using an implementation of the density-based K-Modes clustering algorithm [CLB09] with a distance heuristic designed to optimize cluster separation [NLHH07] available through the `kmodes` package[9] [dV21]. Student profile labels were appended to the student travel simulation results, then used to produce Monte Carlo proportional estimates of profiles by school.

The results in Figure 3 reveal strong dissimilarities in student makeup between schools on the periphery of Knox County and those nearer to Knoxville's downtown core in the center of the county. We estimate that the former are largely composed of students in married families, above poverty, and with employed caregivers, whereas the latter are characterized more strongly by single caregiver living arrangements and, particularly in areas north of the downtown core, economic distress (pop-out map).

Workers (Educators)

We evaluated the results of our K–12 educator simulations with respect to POI occupancy characteristics, as informed by commute and work statistics obtained from the PUMS. Specifically, we used work arrival times associated with each synthetic worker (PUMS *JWAP*) to timestamp the start of each work day, and incremented this by daily hours worked (derived from PUMS *WKHP*) to create a second timestamp for work departure. The estimated departure time assumes that each educator travels to the school for a typical 5-day workweek, and is estimated as $JWAP + \frac{WKHP}{5}$.

7. https://hifld-geoplatform.opendata.arcgis.com
8. https://nces.ed.gov/ccd/files.asp
9. https://pypi.org/project/kmodes

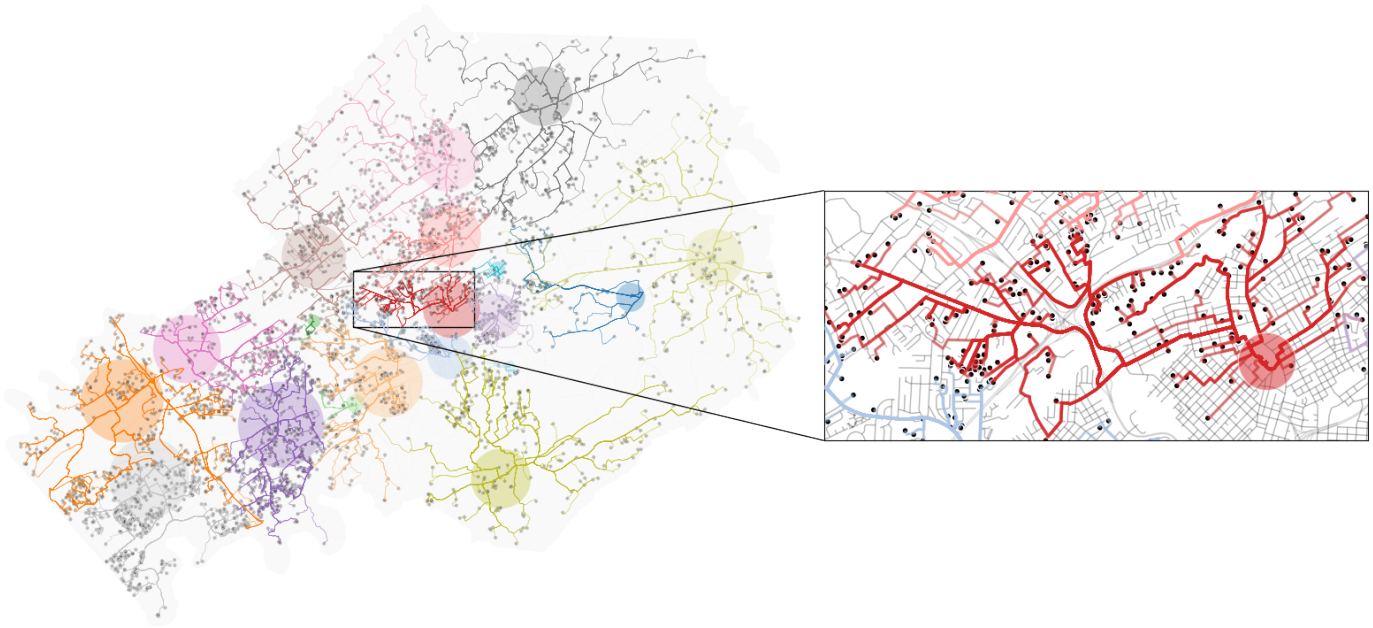


Fig. 2: *Optimal allocations for one simulation of 10^{th} grade public school students in Knox County, TN.*

Fig. 3: *Compositional characteristics of K–12 public schools in Knox County, TN based on 6 student profiles. Glyph plot methodolgy adapted from [GLC+15].*

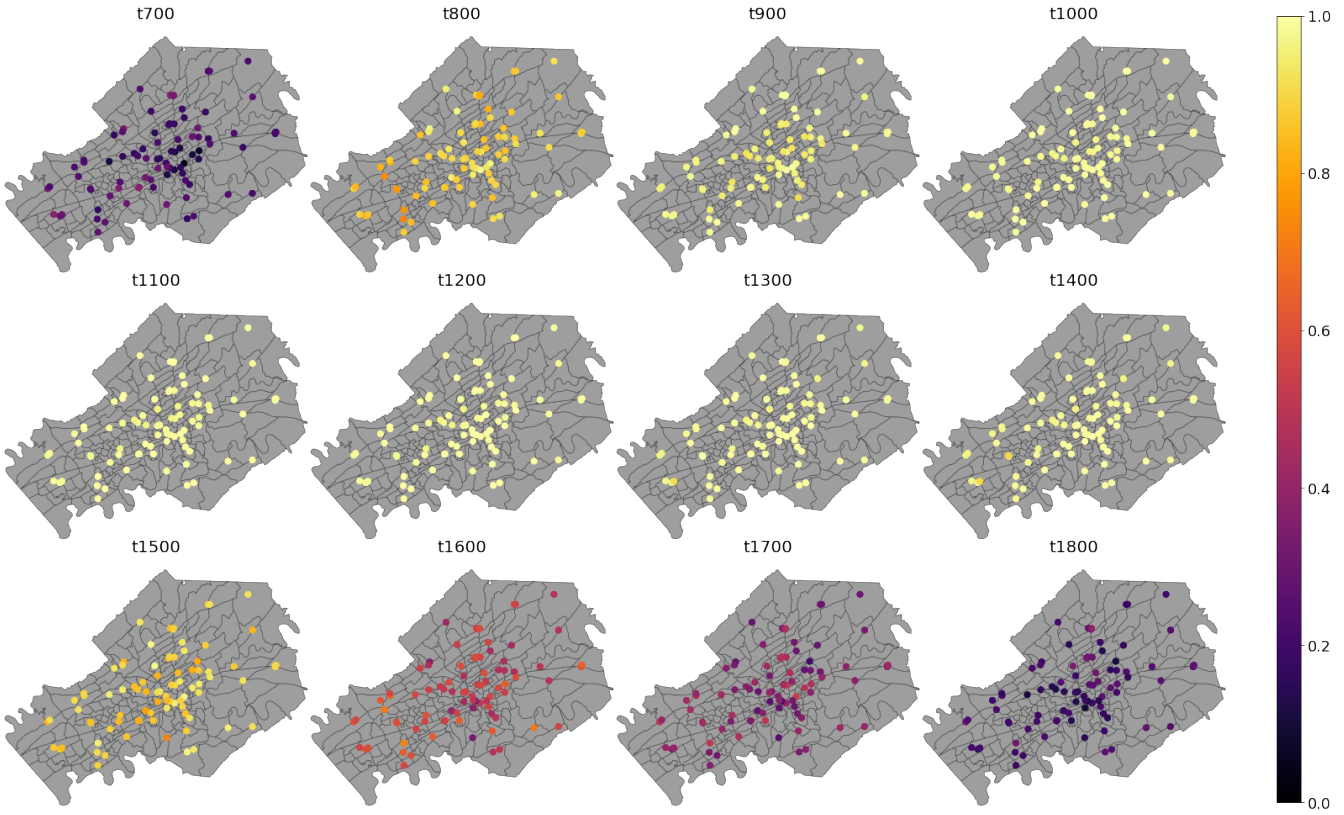


Fig. 4: *Hourly worker occupancy estimates for K–12 schools in Knox County, TN.*

Roughly 50 educator agents per simulation were not attributed with work arrival times, possibly due to the source PUMS respondents being away from their typical workplaces (e.g., on summer or winter break) but still working virtually when they were surveyed. We filled in these unkown arrival times with the modal arrival time observed across all simulations (7:25 AM).

Figure 4 displays the hourly proportion of educators present at each school in Knox County between 7:00 AM (`t700`) and 6:00 PM (`t1800`). Morning worker arrivals occur more rapidly than afternoon departures. Between the hours of 7:00 AM and 9:00 AM (`t700`–`t900`), schools transition from nearly empty of workers to being close to capacity. In the afternoon, workers begin to gradually depart at 3:00 PM (`t1500`) with somewhere between 50%–70% of workers still present by 4:00 PM (`t1600`), then workers begin to depart in earnest at 5:00 PM into 6:00 PM (`t1700`–`t1800`), by which most have returned home.

Geographic differences are also visible and may be a function of (1) a higher concentration of a particular school type (e.g., elementary, middle, high) in this area and (2) staggered starts between these types (to accommodate bus schedules, etc.). This could be due in part to concentrations of different school schedules by grade level, especially elementary schools starting much earlier than middle and high schools[10]. For example, schools near the center of Knox County reach worker capacity more quickly in the morning, starting around 8:00 AM (`t800`), but also empty out more rapidly than schools in surrounding areas beginning around 4:00 PM (`t1600`).

Validation & Diagnostics

A determination of modeling output robustness was needed to validate our results. Specifically, we aimed to ensure the preservation of relative facility size and composition. To perform this validation, we tested the optimal allocations of those generated by Likeness against the maximally adjusted reported enrollment & faculty employment counts. We used the maximum adjusted value to account for scenarios where the population synthesis phase resulted in a total demographic segment greater than reported total facility capacity. We employed Canonical Correlation Analysis (CCA) [Kna78] for the K–12 public school student allocations due to their stratified nature, and an ordinary least squares (OLS) simple linear regression for the educator allocations [PVG+11]. Because CCA is a multivariate measure, it is only a suitable diagnostic for activity allocation when multiple segments (e.g., students by grade) are of interest. For educators, which we treated as a single agent segment without stratification, we used OLS regression instead. The CCA for students was performed in two components: Between-Destination, which measures capacity across *facilities*, and Within-Destination, which measures capacity across *strata*.

Descriptive Monte Carlo statistics from the 30 simulations were run on the resultant coefficients of determination (R^2), which show a goodness of fit (approaching 1). As seen in Table 1, all models performed exceedingly well, though the Within-Destination CCA performed *slightly* less well than both the Between-Destination CCA and the OLS linear regression. In fact, the global minimum of all R^2 scores approaches 0.99 (students – Within-Destination), which demonstrates robust preservation of

10. https://www.knoxschools.org/Page/5553

K–12	R^2 Type	Min	Median	Mean	Max
Students (public schools)	Between-Destination CCA	0.9967	0.9974	0.9973	0.9976
	Within-Destination CCA	0.9883	0.9894	0.9896	0.9910
Educators (public & private schools)	OLS Linear Regression	0.9977	0.9983	0.9983	0.9991

TABLE 1: *Validating optimal allocations considering reported enrollment at public schools & faculty employment at all schools.*

true capacities in our synthetic activity modeling. Furthermore, a global maximum of greater than 0.999 is seen for educators, which indicates a near perfect replication of relative faculty sizes by school.

Discussion

Our Case Study demonstrates the twofold benefits of modeling human dynamics with vivid synthetic populations. Using Likeness, we are able to both produce a more reasoned estimate of the neighborhoods in which people reside and interact than existing synthetic population frameworks, as well as support more nuanced characterization of human activities at specific POIs (e.g., social contact networks, occupancy).

The examples provided in the Case Study show how this refined understanding of human dynamics can benefit planning applications. For example, in the event of a localized emergency, the results of Students could be used to examine schools for which rendezvous with caregivers might pose an added challenge towards students (e.g., more students from single caregiver vs. married family households). Additionally, the POI occupancy dynamics demonstrated in Workers (Educators) could be used to assess the times at which worker commutes to/from places of employment might be most sensitive to a nearby disruption. Another application in the public health sphere might be to use occupancy estimates to anticipate the best time of day to reach workers, during a vaccination campaign, for example.

Our case study had several limitations that we plan to overcome in future work. First, we assumed that all travel within our study area occurs along road networks. While road-based travel *is* the dominant means of travel in the Knoxville CBSA, this assumption is not transferable to other urban areas within the United States. Our eventual goal is to build in additional modes of travel like public transit, walk/bike, and ferries by expanding our ingest of OpenStreetMap features.

Second, we do not yet offer direct support for non-traditional schools (e.g., populations with special needs, families on military bases). For example, the Tennessee School for the Deaf falls within our study area, and its compositional estimate could be refined if we reapportioned students more likely in attendance to that location.

Third, we did not account for teachers in virtual schools, which may form a portion of the missing work arrival times discussed in Workers (Educators). Work-from-home populations can be better incorporated into our travel simulations by applying work schedules from time-use surveys to probabilistically assign in-person or remote status based on occupation. We are particularly interested in using this technique with Likeness to better understand changing patterns of life during the COVID-19 pandemic in 2020.

Conclusion

The Likeness toolkit enhances agent creation for modeling human dynamics through its dual capabilities of high-fidelity ("vivid") agent characterization and travel along real-world transportation networks to POIs. These capabilities benefit planners and urban researchers by providing a richer understanding of how spatial policy interventions can be designed with respect to how people live, move, and interact. Likeness strives to be flexible toward a variety of research applications linked to human security, among them spatial epidemiology, transportation equity, and environmental hazards.

Several ongoing developments will further Likeness' capabilities. First, we plan to expand our support for POIs curated by location services (e.g., Google, Facebook, Here, TomTom, FourSquare) by the ORNL PlanetSense project [TBP+15] by incorporating factors like facility size, hours of operation, and popularity curves to refine the destination capacity estimates required to perform `actlike` simulations. Second, along with multimodal travel, we plan to incorporate multiple trip models based on large-scale human activity datasets like the American Time Use Survey[11] and National Household Travel Survey[12]. Together, these improvements will extend our travel simulations to "non-obligate" population segments traveling to civic, social, and recreational activities [BMWR22]. Third, the current procedure for spatial allocation uses block groups as the target scale for population synthesis. However, there are a limited number of constraining variables available at the block group level. To include a larger volume of constraints (e.g., vehicle access, language), we are exploring an additional tract-level approach. P-MEDM in this case is run on cross-covariances between tracts and "supertract" aggregations created with the Max-*p*-regions problem [DAR12], [WRK21] implemented in PySAL's `spopt` [RA07], [FGK+21], [RAA+21], [FBG+22].

As a final note, the Likeness toolkit is being developed on top of key open source dependencies in the Scientific Python ecosystem, the core of which are, of course, `numpy` [HMvdW+20] and `scipy` [VGO+20]. Although an exhaustive list would be prohibitive, major packages not previously mentioned include `geopandas` [JdBF+21], `matplotlib` [Hun07], `networkx` [HSS08], `pandas` [pdt20], [WM10], and `shapely` [G+]. Our goal is contribute to the community with releases of the packages comprising Likeness, but since this is an emerging project its development to date has been limited to researchers at ORNL. However, we plan to provide a fully open-sourced code base within the coming year through GitHub[13].

Acknowledgements

This material is based upon the work supported by the U.S. Department of Energy under contract no. DE-AC05-00OR22725.

References

[ANM+18] H.M. Abdul Aziz, Nicholas N. Nagle, April M. Morton, Michael R. Hilliard, Devin A. White, and Robert N. Stew-

11. https://www.bls.gov/tus
12. https://nhts.ornl.gov
13. https://github.com/ORNL

art. Exploring the impact of walk–bike infrastructure, safety perception, and built-environment on active transportation mode choice: a random parameter model using New York City commuter data. *Transportation*, 45(5):1207–1229, 2018. doi:10.1007/s11116-017-9760-8.

[BBE+08] Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, and Madhav V. Marathe. EpiSimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008. doi:10.1109/SC.2008.5214892.

[BBM96] Richard J. Beckman, Keith A. Baggerly, and Michael D. McKay. Creating synthetic baseline populations. *Transportation Research Part A: Policy and Practice*, 30(6):415–429, 1996. doi:10.1016/0965-8564(96)00004-3.

[BCD+06] Dimitris Ballas, Graham Clarke, Danny Dorling, Jan Rigby, and Ben Wheeler. Using geographical information systems and spatial microsimulation for the analysis of health inequalities. *Health Informatics Journal*, 12(1):65–79, 2006. doi:10.1177/1460458206061217.

[BFH+17] Komal Basra, M. Patricia Fabian, Raymond R. Holberger, Robert French, and Jonathan I. Levy. Community-engaged modeling of geographic and demographic patterns of multiple public health risk factors. *International Journal of Environmental Research and Public Health*, 14(7):730, 2017. doi:10.3390/ijerph14070730.

[BMWR22] Christa Brelsford, Jessica J. Moehl, Eric M. Weber, and Amy N. Rose. Segmented Population Models: Improving the LandScan USA Non-Obligate Population Estimate (NOPE). American Association of Geographers 2022 Annual Meeting, 2022.

[Boe17] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, September 2017. doi:10.1016/j.compenvurbsys.2017.05.004.

[CGSdG08] Isabel Correia, Luís Gouveia, and Francisco Saldanha-da Gama. Solving the variable size bin packing problem with discretized formulations. *Computers & Operations Research*, 35(6):2103–2113, June 2008. doi:10.1016/j.cor.2006.10.014.

[CLB09] Fuyuan Cao, Jiye Liang, and Liang Bai. A new initialization method for categorical data clustering. *Expert Systems with Applications*, 36(7):10223–10228, 2009. doi:10.1016/j.eswa.2009.01.060.

[DAR12] Juan C. Duque, Luc Anselin, and Sergio J. Rey. THE MAX-P-REGIONS PROBLEM*. *Journal of Regional Science*, 52(3):397–419, 2012. doi:10.1111/j.1467-9787.2011.00743.x.

[DKA+08] M. Diaz, J.J. Kim, G. Albero, S. De Sanjose, G. Clifford, F.X. Bosch, and S.J. Goldie. Health and economic impact of HPV 16 and 18 vaccination and cervical cancer screening in India. *British Journal of Cancer*, 99(2):230–238, 2008. doi:10.1038/sj.bjc.6604462.

[dV21] Nelis J. de Vos. kmodes categorical clustering library. https://github.com/nicodv/kmodes, 2015–2021.

[FBG+22] Xin Feng, Germano Barcelos, James D. Gaboardi, Elijah Knaap, Ran Wei, Levi J. Wolf, Qunshan Zhao, and Sergio J. Rey. spopt: a python package for solving spatial optimization problems in PySAL. *Journal of Open Source Software*, 7(74):3330, 2022. doi:10.21105/joss.03330.

[FGK+21] Xin Feng, James D. Gaboardi, Elijah Knaap, Sergio J. Rey, and Ran Wei. pysal/spopt, jan 2021. URL: https://github.com/pysal/spopt, doi:10.5281/zenodo.4444156.

[FL86] D.K. Friesen and M.A. Langston. Variable Sized Bin Packing. *SIAM Journal on Computing*, 15(1):222–230, February 1986. doi:10.1137/0215016.

[FW12] Fletcher Foti and Paul Waddell. A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale. In *Transportation Research Board Annual Conference*, pages 1–14, 2012. URL: https://onlinepubs.trb.org/onlinepubs/conferences/2012/4thITM/Papers-A/0117-000062.pdf.

[G+] Sean Gillies et al. Shapely: manipulation and analysis of geometric objects, 2007–. URL: https://github.com/shapely/shapely.

[GFH20] James D. Gaboardi, David C. Folch, and Mark W. Horner. Connecting Points to Spatial Networks: Effects on Discrete Optimization Models. *Geographical Analysis*, 52(2):299–322, 2020. doi:10.1111/gean.12211.

[GLC+15] Isabella Gollini, Binbin Lu, Martin Charlton, Christopher Brunsdon, and Paul Harris. GWmodel: An R package for exploring spatial heterogeneity using geographically weighted models. *Journal of Statistical Software*, 63(17):1–50, 2015. doi:10.18637/jss.v063.i17.

[GT22] James D. Gaboardi and Joseph V. Tuccillo. Simulating Travel to Points of Interest for Demographically-rich Synthetic Populations, February 2022. American Association of Geographers Annual Meeting. doi:10.5281/zenodo.6335783.

[Hew97] Kenneth Hewitt. Vulnerability Perspectives: the Human Ecology of Endangerment. In *Regions of Risk: A Geographical Introduction to Disasters*, chapter 6, pages 141–164. Addison Wesley Longman, 1997.

[HHSB12] Kirk Harland, Alison Heppenstall, Dianna Smith, and Mark H. Birkin. Creating realistic synthetic populations at varying spatial scales: A comparative critique of population synthesis techniques. *Journal of Artificial Societies and Social Simulation*, 15(1):1, 2012. doi:10.18564/jasss.1909.

[Hit41] Frank L. Hitchcock. The Distribution of a Product from Several Sources to Numerous Localities. *Journal of Mathematics and Physics*, 20(1-4):224–230, 1941. doi:10.1002/sapm1941201224.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[HNB+11] Jan A.C. Hontelez, Nico Nagelkerke, Till Bärnighausen, Roel Bakker, Frank Tanser, Marie-Louise Newell, Mark N. Lurie, Rob Baltussen, and Sake J. de Vlas. The potential impact of RV144-like vaccines in rural South Africa: a study using the STDSIM microsimulation model. *Vaccine*, 29(36):6100–6106, 2011. doi:10.1016/j.vaccine.2011.06.059.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. URL: https://www.osti.gov/biblio/960616.

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

[JdBF+21] Kelsey Jordahl, Joris Van den Bossche, Martin Fleischmann, James McBride, Jacob Wasserman, Adrian Garcia Badaracco, Jeffrey Gerard, Alan D. Snow, Jeff Tratner, Matthew Perry, Carson Farmer, Geir Arne Hjelle, Micah Cochran, Sean Gillies, Lucas Culbertson, Matt Bartos, Brendan Ward, Giacomo Caria, Mike Taves, Nick Eubank, sangarshanan, John Flavin, Matt Richards, Sergio Rey, maxalbert, Aleksey Bilogur, Christopher Ren, Dani Arribas-Bel, Daniel Mesejo-León, and Leah Wasser. geopandas/geopandas: v0.10.2, October 2021. doi:10.5281/zenodo.5573592.

[Kna78] Thomas R. Knapp. Canonical Correlation Analysis: A general parametric significance-testing system. *Psychological Bulletin*, 85(2):410–416, 1978. doi:10.1037/0033-2909.85.2.410.

[Koo49] Tjalling C. Koopmans. Optimum Utilization of the Transportation System. *Econometrica*, 17:136–146, 1949. Publisher: [Wiley, Econometric Society]. doi:10.2307/1907301.

[LB13] Robin Lovelace and Dimitris Ballas. 'Truncate, replicate, sample': A method for creating integer weights for spatial microsimulation. *Computers, Environment and Urban Systems*, 41:1–11, September 2013. doi:10.1016/j.compenvurbsys.2013.03.004.

[LNB13] Stefan Leyk, Nicholas N. Nagle, and Barbara P. Buttenfield. Maximum Entropy Dasymetric Modeling for Demographic Small Area Estimation. *Geographical Analysis*, 45(3):285–306, July 2013. doi:10.1111/gean.12011.

[MCB+08] Karyn Morrissey, Graham Clarke, Dimitris Ballas, Stephen Hynes, and Cathal O'Donoghue. Examining access to GP services in rural Ireland using microsimulation analysis. *Area*, 40(3):354–364, 2008. doi:10.1111/j.1475-4762.2008.00844.x.

[MNP+17] April M. Morton, Nicholas N. Nagle, Jesse O. Piburn, Robert N. Stewart, and Ryan McManamay. A hybrid dasymetric and machine learning approach to high-resolution residential electricity consumption modeling. In *Advances in Geocomputation*, pages 47–58. Springer, 2017. doi:10.1007/978-3-319-22786-3_5.

[MOD11] Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. PuLP: A Linear Programming Toolkit for Python. Technical report, 2011. URL: https://www.dit.uoi.gr/e-class/modules/document/file.php/216/PAPERS/2011.%20PuLP%20-%20A%20Linear%20Programming%20Toolkit%20for%20Python.pdf.

[MPN+17] April M. Morton, Jesse O. Piburn, Nicholas N. Nagle, H.M. Aziz, Samantha E. Duchscherer, and Robert N. Stewart. A simulation approach for modeling high-resolution daytime commuter travel flows and distributions of worker subpopulations. In *GeoComputation 2017, Leeds, UK*, pages 1–5, 2017. URL: http://www.geocomputation.org/2017/papers/44.pdf.

[MS01] Harvey J. Miller and Shih-Lung Shaw. *Geographic Information Systems for Transportation: Principles and Applications*. Oxford University Press, New York, 2001.

[MS15] Harvey J. Miller and Shih-Lung Shaw. Geographic Information Systems for Transportation in the 21st Century. *Geography Compass*, 9(4):180–189, 2015. doi:10.1111/gec3.12204.

[NBLS14] Nicholas N. Nagle, Barbara P. Buttenfield, Stefan Leyk, and Seth Spielman. Dasymetric modeling and uncertainty. *Annals of the Association of American Geographers*, 104(1):80–95, 2014. doi:10.1080/00045608.2013.843439.

[NCA13] Markku Nurhonen, Allen C. Cheng, and Kari Auranen. Pneumococcal transmission and disease in silico: a microsimulation model of the indirect effects of vaccination. *PloS one*, 8(2):e56079, 2013. doi:10.1371/journal.pone.0056079.

[NLHH07] Michael K. Ng, Mark Junjie Li, Joshua Zhexue Huang, and Zengyou He. On the impact of dissimilarity measure in k-modes clustering algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(3):503–507, 2007. doi:10.1109/TPAMI.2007.53.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. doi:10.5281/zenodo.3509134.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL: https://www.jmlr.org/papers/v12/pedregosa11a.html.

[QC13] Fang Qiu and Robert Cromley. Areal Interpolation and Dasymetric Modeling: Areal Interpolation and Dasymetric Modeling. *Geographical Analysis*, 45(3):213–215, July 2013. doi:10.1111/gean.12016.

[RA07] Sergio J. Rey and Luc Anselin. PySAL: A Python Library of Spatial Analytical Methods. *The Review of Regional Studies*, 37(1):5–27, 2007. URL: https://rrs.scholasticahq.com/article/8285.pdf, doi:10.52324/001c.8285.

[RAA+21] Sergio J. Rey, Luc Anselin, Pedro Amaral, Dani Arribas-Bel, Renan Xavier Cortes, James David Gaboardi, Wei Kang, Elijah Knaap, Ziqi Li, Stefanie Lumnitz, Taylor M. Oshan, Hu Shao, and Levi John Wolf. The PySAL Ecosystem: Philosophy and Implementation. *Geographical Analysis*, 2021. doi:10.1111/gean.12276.

[RSF+21] Krishna P. Reddy, Fatma M. Shebl, Julia H.A. Foote, Guy Harling, Justine A. Scott, Christopher Panella, Kieran P. Fitzmaurice, Clare Flanagan, Emily P. Hyle, Anne M. Neilan, et al. Cost-effectiveness of public health strategies for COVID-19 epidemic control in South Africa: a microsimulation modelling study. *The Lancet Global Health*, 9(2):e120–e129, 2021. doi:10.1016/S2214-109X(20)30452-6.

[RWM+17] Amy N. Rose, Eric M. Weber, Jessica J. Moehl, Melanie L. Laverdiere, Hsiu-Han Yang, Matthew C. Whitehead, Kelly M. Sims, Nathan E. Trombley, and Budhendra L. Bhaduri. LandScan USA 2016 [Data set]. Technical report, Oak Ridge National Laboratory, 2017. doi:10.48690/1523377.

[SEM14] Samarth Swarup, Stephen G. Eubank, and Madhav V. Marathe. Computational epidemiology as a challenge domain for multi-agent systems. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1173–1176, 2014. URL: https://www.ifaamas.org/AAMAS/aamas2014/proceedings/aamas/p1173.pdf.

[SNGJ+09] Beate Sander, Azhar Nizam, Louis P. Garrison Jr., Maarten J. Postma, M. Elizabeth Halloran, and Ira M. Longini Jr. Economic evaluation of influenza pandemic mitigation strategies in the United States using a stochastic microsimulation transmission model. *Value in Health*, 12(2):226–233, 2009. doi:10.1111/j.1524-4733.2008.00437.x.

[SPH11] Dianna M. Smith, Jamie R. Pearce, and Kirk Harland. Can a deterministic spatial microsimulation model provide reliable small-area estimates of health behaviours? An example of smoking prevalence in New Zealand. *Health & Place*, 17(2):618–624, 2011. doi:10.1016/j.healthplace.2011.01.001.

[ST20] Haroldo G. Santos and Túlio A.M. Toffolo. Mixed Integer Linear Programming with Python. Technical report, 2020. URL: https://python-mip.readthedocs.io/_/downloads/en/latest/pdf/.

[TBP+15] Gautam S. Thakur, Budhendra L. Bhaduri, Jesse O. Piburn, Kelly M. Sims, Robert N. Stewart, and Marie L. Urban. PlanetSense: a real-time streaming and spatio-temporal analytics platform for gathering geo-spatial intelligence from open source data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–4, 2015. doi:10.1145/2820783.2820882.

[TCR08] Melanie N. Tomintz, Graham P. Clarke, and Janette E. Rigby. The geography of smoking in Leeds: estimating individual smoking rates and the implications for the location of stop smoking services. *Area*, 40(3):341–353, 2008. doi:10.1111/j.1475-4762.2008.00837.x.

[TG22] Joseph V. Tuccillo and James D. Gaboardi. Connecting Vivid Population Data to Human Dynamics, June 2022. Distilling Diversity by Tapping High-Resolution Population and Survey Data. doi:10.5281/zenodo.6607533.

[TM21] Joseph V. Tuccillo and Jessica Moehl. An Individual-Oriented Typology of Social Areas in the United States, May 2021. 2021 ACS Data Users Conference. doi:10.5281/zenodo.6672291.

[TMKD17] Matthias Templ, Bernhard Meindl, Alexander Kowarik, and Olivier Dupriez. Simulation of synthetic complex data: The R package simPop. *Journal of Statistical Software*, 79:1–38, 2017. doi:10.18637/jss.v079.i10.

[Tuc21] Joseph V. Tuccillo. An Individual-Centered Approach for Geodemographic Classification. In *11th International Conference on Geographic Information Science 2021 Short Paper Proceedings*, pages 1–6, 2021. doi:10.25436/E2H59M.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C.J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[WCC+09] William D. Wheaton, James C. Cajka, Bernadette M. Chasteen, Diane K. Wagener, Philip C. Cooley, Laxminarayana Ganapathi, Douglas J. Roberts, and Justine L. Allpress. Synthesized population databases: A US geospatial database for agent-based models. *Methods report (RTI Press)*, 2009(10):905, 2009. doi:10.3768/rtipress.2009.mr.0010.0905.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a.

[WRK21] Ran Wei, Sergio J. Rey, and Elijah Knaap. Efficient re-

gionalization for spatially explicit neighborhood delineation. *International Journal of Geographical Information Science*, 35(1):135–151, 2021. doi:10.1080/13658816.2020.1759806.

[ZFJ14] Yi Zhu and Joseph Ferreira Jr. Synthetic population generation at disaggregated spatial scales for land use and transportation microsimulation. *Transportation Research Record*, 2429(1):168–177, 2014. doi:10.3141/2429-18.

poliastro: a Python library for interactive astrodynamics

Juan Luis Cano Rodríguez‡*, Jorge Martínez Garrido‡

https://www.youtube.com/watch?v=VCpTgU1pb5k

Abstract—Space is more popular than ever, with the growing public awareness of interplanetary scientific missions, as well as the increasingly large number of satellite companies planning to deploy satellite constellations. Python has become a fundamental technology in the astronomical sciences, and it has also caught the attention of the Space Engineering community.

One of the requirements for designing a space mission is studying the trajectories of satellites, probes, and other artificial objects, usually ignoring non-gravitational forces or treating them as perturbations: the so-called n-body problem. However, for preliminary design studies and most practical purposes, it is sufficient to consider only two bodies: the object under study and its attractor.

Even though the two-body problem has many analytical solutions, orbit propagation (the initial value problem) and targeting (the boundary value problem) remain computationally intensive because of long propagation times, tight tolerances, and vast solution spaces. On the other hand, astrodynamics researchers often do not share the source code they used to run analyses and simulations, which makes it challenging to try out new solutions.

This paper presents poliastro, an open-source Python library for interactive astrodynamics that features an easy-to-use API and tools for quick visualization. poliastro implements core astrodynamics algorithms (such as the resolution of the Kepler and Lambert problems) and leverages numba, a Just-in-Time compiler for scientific Python, to optimize the running time. Thanks to Astropy, poliastro can perform seamless coordinate frame conversions and use proper physical units and timescales. At the moment, poliastro is the longest-lived Python library for astrodynamics, has contributors from all around the world, and several New Space companies and people in academia use it.

Index Terms—astrodynamics, orbital mechanics, orbit propagation, orbit visualization, two-body problem

Introduction

History

The term "astrodynamics" was coined by the American astronomer Samuel Herrick, who received encouragement from the space pioneer Robert H. Goddard, and refers to the branch of space science dealing with the motion of artificial celestial bodies ([Dub73], [Her71]). However, the roots of its mathematical foundations go back several centuries.

Kepler first introduced his laws of planetary motion in 1609 and 1619 and derived his famous transcendental equation (1), which we now see as capturing a restricted form of the two-body problem. This work was generalized by Newton to give birth to the n-body problem, and many other mathematicians worked on it throughout the centuries (Daniel and Johann Bernoulli, Euler, Gauss). Poincaré established in the 1890s that no general closed-form solution exists for the n-body problem, since the resulting dynamical system is chaotic [Bat99]. Sundman proved in the 1900s the existence of convergent solutions for a few restricted with $n = 3$.

$$M = E - e \sin E \tag{1}$$

In 1903 Tsiokovsky evaluated the conditions required for artificial objects to leave the orbit of the earth; this is considered as a foundational contribution to the field of astrodynamics. Tsiokovsky devised equation 2 which relates the increase in velocity with the effective exhaust velocity of thrusted gases and the fraction of used propellant.

$$\Delta v = v_e \ln \frac{m_0}{m_f} \tag{2}$$

Further developments by Kondratyuk, Hohmann, and Oberth in the early 20th century all added to the growing field of orbital mechanics, which in turn enabled the development of space flight in the USSR and the United States in the 1950s and 1960s.

The two-body problem

In a system of $i \in 1, ..., n$ bodies subject to their mutual attraction, by application of Newton's law of universal gravitation, the total force $\mathbf{f}_i$ affecting m_i due to the presence of the other $n-1$ masses is given by [Bat99]:

$$\mathbf{f}_i = -G \sum_{j \neq i}^{n} \frac{m_i m_j}{|\mathbf{r}_{ij}|^3} \mathbf{r}_{ij} \tag{3}$$

where $G = 6.67430 \cdot 10^{-11}$ N m^2 kg^{-2} is the universal gravitational constant, and $\mathbf{r}_{ij}$ denotes the position vector from m_i to m_j. Applying Newton's second law of motion results in a system of n differential equations:

$$\frac{\mathrm{d}^2 \mathbf{r}_i}{\mathrm{d}t^2} = -G \sum_{j \neq i}^{n} \frac{m_j}{|\mathbf{r}_{ij}|^3} \mathbf{r}_{ij} \tag{4}$$

By setting $n = 2$ in 4 and subtracting the two resulting equalities, one arrives to the **fundamental equation of the two-body problem**:

$$\frac{\mathrm{d}^2 \mathbf{r}}{\mathrm{d}t^2} = -\frac{\mu}{r^3} \mathbf{r} \tag{5}$$

where $\mu = G(m_1 + m_2) = G(M + m)$. When $m \ll M$ (for example, an artificial satellite orbiting a planet), one can consider $\mu = GM$ a property of the attractor.

* *Corresponding author: hello@juanlu.space*
‡ *Unaffiliated*

Keplerian vs non-keplerian motion

Conveniently manipulating equation 5 leads to several properties [Bat99] that were already published by Johannes Kepler in the 1610s, namely:

1) The orbit always describes a conic section (an ellipse, a parabola, or an hyperbola), with the attractor at one of the two foci and can be written in polar coordinates like $r = \frac{p}{1+e\cos\nu}$ (Kepler's first law).
2) The magnitude of the specific angular momentum $h = r^2 \frac{\mathrm{d}\theta}{\mathrm{d}t}$ is constant an equal to two times the areal velocity (Kepler's second law).
3) For closed (circular and elliptical) orbits, the period is related to the size of the orbit through $P = 2\pi\sqrt{\frac{a^3}{\mu}}$ (Kepler's third law).

For many practical purposes it is usually sufficient to limit the study to one object orbiting an attractor and ignore all other external forces of the system, hence restricting the study to trajectories governed by equation 5. Such trajectories are called "Keplerian", and several problems can be formulated for them:

- The **initial-value problem**, which is usually called **propagation**, involves determining the position and velocity of an object after an elapse period of time given some initial conditions.
- **Preliminary orbit determination**, which involves using exact or approximate methods to derive a Keplerian orbit from a set of observations.
- The **boundary-value problem**, often named **the Lambert problem**, which involves determining a Keplerian orbit from boundary conditions, usually departure and arrival position vectors and a time of flight.

Fortunately, most of these problems boil down to finding numerical solutions to relatively simple algebraic relations between time and angular variables: for elliptic motion ($0 \leq e < 1$) it is the Kepler equation, and equivalent relations exist for the other eccentricity regimes [Bat99]. Numerical solutions for these equations can be found in a number of different ways, each one with different complexity and precision tradeoffs. In the Methods section we list the ones implemented by poliastro.

On the other hand, there are many situations in which natural and artificial orbital perturbations must be taken into account so that the actual non-Keplerian motion can be properly analyzed:

- Interplanetary travel in the proximity of other planets. On a first approximation it is usually enough to study the trajectory in segments and focus the analysis on the closest attractor, hence patching several Keplerian orbits along the way (the so-called "patched-conic approximation") [Bat99]. The boundary surface that separates one segment from the other is called the sphere of influence.
- Use of solar sails, electric propulsion, or other means of continuous thrust. Devising the optimal guidance laws that minimize travel time or fuel consumption under these conditions is usually treated as an optimization problem of a dynamical system, and as such it is particularly challenging [Con14].
- Artificial satellites in the vicinity of a planet. This is the regime in which all the commercial space industry operates, especially for those satellites in Low-Earth Orbit (LEO).

State of the art

In our view, at the time of creating poliastro there were a number of issues with existing open source astrodynamics software that posed a barrier of entry for novices and amateur practitioners. Most of these barriers still exist today and are described in the following paragraphs. The goals of the project can be condensed as follows:

1) Set an example on reproducibility and good coding practices in astrodynamics.
2) Become an approachable software even for novices.
3) Offer a performant software that can be also used in scripting and interactive workflows.

The most mature software libraries for astrodynamics are arguably Orekit [noa22c], a "low level space dynamics library written in Java" with an open governance model, and SPICE [noa22d], a toolkit developed by NASA's Navigation and Ancillary Information Facility at the Jet Propulsion Laboratory. Other similar, smaller projects that appeared later on and that are still maintained to this day include PyKEP [IBD+20], beyond [noa22a], tudatpy [noa22e], sbpy [MKDVB+19], Skyfield [Rho20] (Python), CelestLab (Scilab) [noa22b], astrodynamics.jl (Julia) [noa] and Nyx (Rust) [noa21a]. In addition, there are some Graphical User Interface (GUI) based open source programs used for Mission Analysis and orbit visualization, such as GMAT [noa20] and gpredict [noa18], and complete web applications for tracking constellations of satellites like the SatNOGS project by the Libre Space Foundation [noa21b].

The level of quality and maintenance of these packages is somewhat heterogeneous. Community-led projects with a strong corporate backing like Orekit are in excellent health, while on the other hand smaller projects developed by volunteers (beyond, astrodynamics.jl) or with limited institutional support (PyKEP, GMAT) suffer from lack of maintenance. Part of the problem might stem from the fact that most scientists are never taught how to build software efficiently, let alone the skills to collaboratively develop software in the open [WAB+14], and astrodynamicists are no exception.

On the other hand, it is often difficult to translate the advances in astrodynamics research to software. Classical algorithms developed throughout the 20th century are described in papers that are sometimes difficult to find, and source code or validation data is almost never available. When it comes to modern research carried in the digital era, source code and validation data is still difficult, even though they are supposedly provided "upon reasonable request" [SSM18] [GBP22].

It is no surprise that astrodynamics software often requires deep expertise. However, there are often implicit assumptions that are not documented with an adequate level of detail which originate widespread misconceptions and lead even seasoned professionals to make conceptual mistakes. Some of the most notorious misconceptions arise around the use of general perturbations data (OMMs and TLEs) [Fin07], the geometric interpretation of the mean anomaly [Bat99], or coordinate transformations [VCHK06].

Finally, few of the open source software libraries mentioned above are amenable to scripting or interactive use, as promoted by computational notebooks like Jupyter [KRKP+16].

The following sections will now discuss the various areas of current research that an astrodynamicist will engage in, and how poliastro improves their workflow.

Methods

Software Architecture

The architecture of poliastro emerges from the following set of conflicting requirements:

1) There should be a high-level API that enables users to perform orbital calculations in a straightforward way and prevent typical mistakes.
2) The running time of the algorithms should be within the same order of magnitude of existing compiled implementations.
3) The library should be written in a popular open-source language to maximize adoption and lower the barrier to external contributors.

One of the most typical mistakes we set ourselves to prevent with the high-level API is dimensional errors. Addition and substraction operations of physical quantities are defined only for quantities with the same units [Dro53]: for example, the operation $1\ \mathrm{km} + 100\ \mathrm{m}$ requires a scale transformation of at least one of the operands, since they have different units (kilometers and meters) but the same dimension (length), whereas the operation $1\ \mathrm{km} + 1\ \mathrm{kg}$ is directly not allowed because dimensions are incompatible (length and mass). As such, software systems operating with physical quantities should raise exceptions when adding different dimensions, and transparently perform the required scale transformations when adding different units of the same dimension.

With this in mind, we evaluated several Python packages for unit handling (see [JGAZJT+18] for a recent survey) and chose `astropy.units` [TPWS+18].

```
radius = 6000  # km
altitude = 500  # m

# Wrong!
distance = radius + altitude

from astropy import units as u

# Correct
distance = (radius << u.km) + (altitude << u.m)
```

This notion of providing a "safe" API extends to other parts of the library by leveraging other capabilities of the Astropy project. For example, timestamps use `astropy.time` objects, which take care of the appropriate handling of time scales (such as TDB or UTC), reference frame conversions leverage `astropy.coordinates`, and so forth.

One of the drawbacks of existing unit packages is that they impose a significant performance penalty. Even though `astropy.units` is integrated with NumPy, hence allowing the creation of array quantities, all the unit compatibility checks are implemented in Python and require lots of introspection, and this can slow down mathematical operations by several orders of magnitude. As such, to fulfill our desired performance requirement for poliastro, we envisioned a two-layer architecture:

- The **Core API** follows a procedural style, and all the functions receive Python numerical types and NumPy arrays for maximum performance.
- The **High level API** is object-oriented, all the methods receive Astropy `Quantity` objects with physical units, and computations are deferred to the Core API.

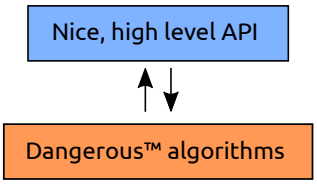


Fig. 1: *poliastro two-layer architecture*

Most of the methods of the High level API consist only of the necessary unit compatibility checks, plus a wrapper over the corresponding Core API function that performs the actual computation.

```
@u.quantity_input(E=u.rad, ecc=u.one)
def E_to_nu(E, ecc):
    """True anomaly from eccentric anomaly."""
    return (
        E_to_nu_fast(
            E.to_value(u.rad),
            ecc.value
        ) << u.rad
    ).to(E.unit)
```

As a result, poliastro offers a unit-safe API that performs the least amount of computation possible to minimize the performance penalty of unit checks, and also a unit-unsafe API that offers maximum performance at the cost of not performing any unit validation checks.

Finally, there are several options to write performant code that can be used from Python, and one of them is using a fast, compiled language for the CPU intensive parts. Successful examples of this include NumPy, written in C [HMvdW+20], SciPy, featuring a mix of FORTRAN, C, and C++ code [VGO+20], and pandas, making heavy use of Cython [BBC+11]. However, having to write code in two different languages hinders the development speed, makes debugging more difficult, and narrows the potential contributor base (what Julia creators called "The Two Language Problem" [BEKS17]).

As authors of poliastro we wanted to use Python as the sole programming language of the implementation, and the best solution we found to improve its performance was to use Numba, a LLVM-based Python JIT compiler [LPS15].

Usage

Basic `Orbit` and `Ephem` creation

The two central objects of the poliastro high level API are `Orbit` and `Ephem`:

- `Orbit` objects represent an osculating (hence Keplerian) orbit of a dimensionless object around an attractor at a given point in time and a certain reference frame.
- `Ephem` objects represent an ephemerides, a sequence of spatial coordinates over a period of time in a certain reference frame.

There are six parameters that uniquely determine a Keplerian orbit, plus the gravitational parameter of the corresponding attractor (k or μ). Optionally, an epoch that contextualizes the orbit can be included as well. This set of six parameters is not unique, and several of them have been developed over the years to serve different purposes. The most widely used ones are:

- **Cartesian elements**: Three components for the position (x, y, z) and three components for the velocity (v_x, v_y, v_z). This set has no singularities.

- **Classical Keplerian elements**: Two components for the shape of the conic (usually the semimajor axis a or semiparameter p and the eccentricity e), three Euler angles for the orientation of the orbital plane in space (inclination i, right ascension of the ascending node Ω, and argument of periapsis ω), and one polar angle for the position of the body along the conic (usually true anomaly f or ν). This set of elements has an easy geometrical interpretation and the advantage that, in pure two-body motion, five of them are fixed $(a, e, i, \Omega, \omega)$ and only one is time-dependent (ν), which greatly simplifies the analytical treatment of orbital perturbations. However, they suffer from singularities steming from the Euler angles ("gimbal lock") and equations expressed in them are ill-conditioned near such singularities.
- **Walker modified equinoctial elements**: Six parameters (p, f, g, h, k, L). Only L is time-dependent and this set has no singularities, however the geometrical interpretation of the rest of the elements is lost [WIO85].

Here is how to create an `Orbit` from cartesian and from classical Keplerian elements. Walker modified equinoctial elements are supported as well.

```
from astropy import units as u

from poliastro.bodies import Earth, Sun
from poliastro.twobody import Orbit
from poliastro.constants import J2000

# Data from Curtis, example 4.3
r = [-6045, -3490, 2500] << u.km
v = [-3.457, 6.618, 2.533] << u.km / u.s

orb_curtis = Orbit.from_vectors(
    Earth,  # Attractor
    r, v  # Elements
)

# Data for Mars at J2000 from JPL HORIZONS
a = 1.523679 << u.au
ecc = 0.093315 << u.one
inc = 1.85 << u.deg
raan = 49.562 << u.deg
argp = 286.537 << u.deg
nu = 23.33 << u.deg

orb_mars = Orbit.from_classical(
    Sun,
    a, ecc, inc, raan, argp, nu,
    J2000  # Epoch
)
```

When displayed on an interactive REPL, `Orbit` objects provide basic information about the geometry, the attractor, and the epoch:

```
>>> orb_curtis
7283 x 10293 km x 153.2 deg (GCRS) orbit
around Earth (X) at epoch J2000.000 (TT)

>>> orb_mars
1 x 2 AU x 1.9 deg (HCRS) orbit
around Sun (X) at epoch J2000.000 (TT)
```

Similarly, `Ephem` objects can be created using a variety of classmethods as well. Thanks to `astropy.coordinates` built-in low-fidelity ephemerides, as well as its capability to remotely access the JPL HORIZONS system, the user can seamlessly build an object that contains the time history of the position of any Solar System body:

```
from astropy.time import Time
from astropy.coordinates import solar_system_ephemeris
from poliastro.ephem import Ephem

# Configure high fidelity ephemerides globally
# (requires network access)
solar_system_ephemeris.set("jpl")

# For predefined poliastro attractors
earth = Ephem.from_body(Earth, Time.now().tdb)

# For the rest of the Solar System bodies
ceres = Ephem.from_horizons("Ceres", Time.now().tdb)
```

There are some crucial differences between `Orbit` and `Ephem` objects:

- `Orbit` objects have an attractor, whereas `Ephem` objects do not. Ephemerides can originate from complex trajectories that don't necessarily conform to the ideal two-body problem.
- `Orbit` objects capture a precise instant in a two-body motion plus the necessary information to propagate it forward in time indefinitely, whereas `Ephem` objects represent a bounded time history of a trajectory. This is because the equations for the two-body motion are known, whereas an ephemeris is either an observation or a prediction that cannot be extrapolated in any case without external knowledge. As such, `Orbit` objects have a `.propagate` method, but `Ephem` ones do not. This prevents users from attempting to propagate the position of the planets, which will always yield poor results compared to the excellent ephemerides calculated by external entities.

Finally, both types have methods to convert between them:

- `Ephem.from_orbit` is the equivalent of sampling a two-body motion over a given time interval. As explained above, the resulting `Ephem` loses the information about the original attractor.
- `Orbit.from_ephem` is the equivalent of calculating the osculating orbit at a certain point of a trajectory, assuming a given attractor. The resulting `Orbit` loses the information about the original, potentially complex trajectory.

Orbit propagation

`Orbit` objects have a `.propagate` method that takes an elapsed time and returns another `Orbit` with new orbital elements and an updated epoch:

```
>>> from poliastro.examples import iss

>>> iss
>>> 6772 x 6790 km x 51.6 deg (GCRS) ...

>>> iss.nu.to(u.deg)
<Quantity 46.59580468 deg>

>>> iss_30m = iss.propagate(30 << u.min)

>>> (iss_30m.epoch - iss.epoch).datetime
datetime.timedelta(seconds=1800)

>>> (iss_30m.nu - iss.nu).to(u.deg)
<Quantity 116.54513153 deg>
```

The default propagation algorithm is an analytical procedure described in [FCM13] that works seamlessly in the near parabolic region. In addition, poliastro implements analytical propagation algorithms as described in [DB83], [OG86], [Mar95], [Mik87], [PP13], [Cha22], and [VM07].

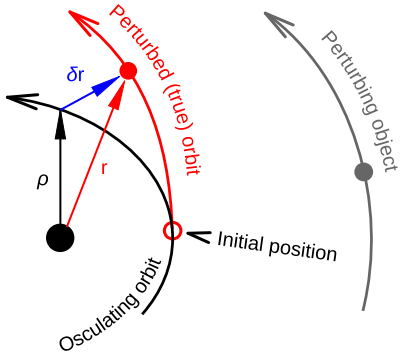


Fig. 2: *Osculating (Keplerian) vs perturbed (true) orbit (source: Wikipedia, CC BY-SA 3.0)*

Natural perturbations

As showcased in Figure 2, at any point in a trajectory we can define an ideal Keplerian orbit with the same position and velocity under the attraction of a point mass: this is called the osculating orbit. Some numerical propagation methods exist that model the true, perturbed orbit as a deviation from an evolving, osculating orbit. poliastro implements Cowell's method [CC10], which consists in adding all the perturbation accelerations and then integrating the resulting differential equation with any numerical method of choice:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\frac{\mu}{r^3}\mathbf{r} + \mathbf{a}_d \tag{6}$$

The resulting equation is usually integrated using high order numerical methods, since the integration times are quite large and the tolerances comparatively tight. An in-depth discussion of such methods can be found in [HNW09]. poliastro uses Dormand-Prince 8(5,3) (`DOP853`), a commonly used method available in SciPy [HMvdW+20].

There are several natural perturbations included: J2 and J3 gravitational terms, several atmospheric drag models (exponential, [Jac77], [AAAA62], [AAA+76]), and helpers for third body gravitational attraction and radiation pressure as described in [**?**].

```
@njit
def combined_a_d(
    t0, state, k, j2, r_eq, c_d, a_over_m, h0, rho0
):
    return (
        J2_perturbation(
            t0, state, k, j2, r_eq
        ) + atmospheric_drag_exponential(
            t0, state, k, r_eq, c_d, a_over_m, h0, rho0
        )
    )

def f(t0, state, k):
    du_kep = func_twobody(t0, state, k)
    ax, ay, az = combined_a_d(
        t0,
        state,
        k,
        R=R,
        C_D=C_D,
        A_over_m=A_over_m,
        H0=H0,
        rho0=rho0,
        J2=Earth.J2.value,
    )
    du_ad = np.array([0, 0, 0, ax, ay, az])

    return du_kep + du_ad
```

```
rr = propagate(
    orbit,
    tofs,
    method=cowell,
    f=f,
)
```

Continuous thrust control laws

Beyond natural perturbations, spacecraft can modify their trajectory on purpose by using impulsive maneuvers (as explained in the next section) as well as continuous thrust guidance laws. The user can define custom guidance laws by providing a perturbation acceleration in the same way natural perturbations are used. In addition, poliastro includes several analytical solutions for continuous thrust guidance laws with specific purposes, as studied in [CR17]: optimal transfer between circular coplanar orbits [Ede61] [Bur67], optimal transfer between circular inclined orbits [Ede61] [Kec97], quasi-optimal eccentricity-only change [Pol97], simultaneous eccentricity and inclination change [Pol00], and agument of periapsis adjustment [Pol98]. A much more rigorous analysis of a similar set of laws can be found in [DCV21].

```
from poliastro.twobody.thrust import change_ecc_inc

ecc_f = 0.0 << u.one
inc_f = 20.0 << u.deg
f = 2.4e-6 << (u.km / u.s**2)

a_d, _, t_f = change_ecc_inc(orbit, ecc_f, inc_f, f)
```

Impulsive maneuvers

Impulsive maneuvers are modeled considering a change in the velocity of a spacecraft while its position remains fixed. The `poliastro.maneuver.Maneuver` class provides various constructors to instantiate popular impulsive maneuvers in the framework of the non-perturbed two-body problem:

- `Maneuver.impulse`
- `Maneuver.hohmann`
- `Maneuver.bielliptic`
- `Maneuver.lambert`

```
from poliastro.maneuver import Maneuver

orb_i = Orbit.circular(Earth, alt=700 << u.km)
hoh = Maneuver.hohmann(orb_i, r_f=36000 << u.km)
```

Once instantiated, `Maneuver` objects provide information regarding total Δv and Δt:

```
>>> hoh.get_total_cost()
<Quantity 3.6173981270031357 km / s>

>>> hoh.get_total_time()
<Quantity 15729.741535747102 s>
```

`Maneuver` objects can be applied to `Orbit` instances using the `apply_maneuver` method.

```
>>> orb_i
7078 x 7078 km x 0.0 deg (GCRS) orbit
around Earth (X)

>>> orb_f = orb_i.apply_maneuver(hoh)
>>> orb_f
36000 x 36000 km x 0.0 deg (GCRS) orbit
around Earth (X)
```

Targeting

Targeting is the problem of finding the orbit connecting two positions over a finite amount of time. Within the context of the non-perturbed two-body problem, targeting is just a matter of solving the BVP, also known as Lambert's problem. Because targeting tries to find for an orbit, the problem is included in the Initial Orbit Determination field.

The `poliastro.iod` package contains `izzo` and `vallado` modules. These provide a `lambert` function for solving the targeting problem. Nevertheless, a `Maneuver.lambert` constructor is also provided so users can keep taking advantage of `Orbit` objects.

```
# Declare departure and arrival datetimes
date_launch = time.Time(
    '2011-11-26 15:02', scale='tdb'
)
date_arrival = time.Time(
    '2012-08-06 05:17', scale='tdb'
)

# Define initial and final orbits
orb_earth = Orbit.from_ephem(
    Sun, Ephem.from_body(Earth, date_launch),
    date_launch
)
orb_mars = Orbit.from_ephem(
    Sun, Ephem.from_body(Mars, date_arrival),
    date_arrival
)

# Compute targetting maneuver and apply it
man_lambert = Maneuver.lambert(orb_earth, orb_mars)
orb_trans, orb_target = ss0.apply_maneuver(
    man_lambert, intermediate=true
)
```

Targeting is closely related to quick mission design by means of porkchop diagrams. These are contour plots showing all combinations of departure and arrival dates with the specific energy for each transfer orbit. They allow for quick identification of the most optimal transfer dates between two bodies.

The `poliastro.plotting.porkchop` provides the `PorkchopPlotter` class which allows the user to generate these diagrams.

```
from poliastro.plotting.porkchop import (
    PorkchopPlotter
)
from poliastro.utils import time_range

# Generate all launch and arrival dates
launch_span = time_range(
    "2020-03-01", end="2020-10-01", periods=int(150)
)
arrival_span = time_range(
    "2020-10-01", end="2021-05-01", periods=int(150)
)

# Create an instance of the porkchop and plot it
porkchop = PorkchopPlotter(
    Earth, Mars, launch_span, arrival_span,
)
```

Previous code, with some additional customization, generates figure 3.

Plotting

For visualization purposes, poliastro provides the `poliastro.plotting` package, which contains various utilities for generating 2D and 3D graphics using different backends such as matplotlib [Hun07] and Plotly [Inc15].

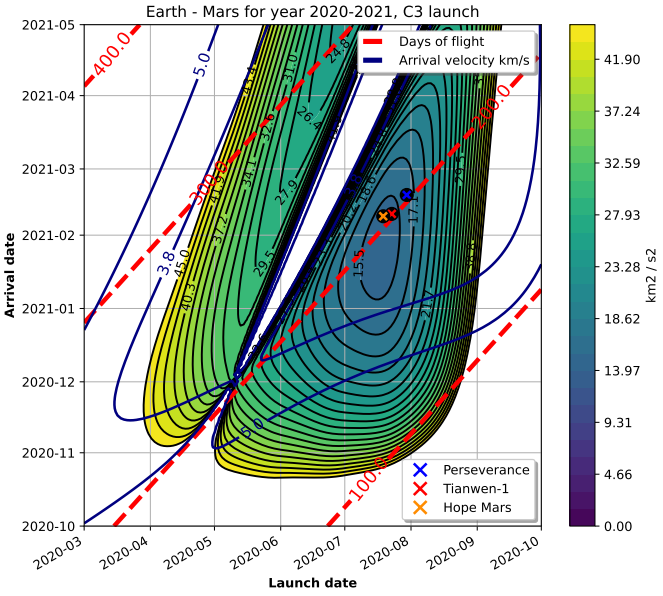


Fig. 3: *Porkchop plot for Earth-Mars transfer arrival energy showing latest missions to the Martian planet.*

Generated graphics can be static or interactive. The main difference between these two is the ability to modify the camera view in a dynamic way when using interactive plotters.

The most important classes in the `poliastro.plotting` package are `StaticOrbitPlotter` and `OrbitPlotter3D`. In addition, the `poliastro.plotting.misc` module contains the `plot_solar_system` function, which allows the user to visualize inner and outter both in 2D and 3D, as requested by users.

The following example illustrates the plotting capabilities of poliastro. At first, orbits to be plotted are computed and their plotting style is declared:

```
from poliastro.plotting.misc import plot_solar_system

# Current datetime
now = Time.now().tdb

# Obtain Florence and Halley orbits
florence = Orbit.from_sbdb("Florence")
halley_1835_ephem = Ephem.from_horizons(
    "90000031", now
)
halley_1835 = Orbit.from_ephem(
    Sun, halley_1835_ephem, halley_1835_ephem.epochs[0]
)

# Define orbit labels and color style
florence_style = {label: "Florence", color: "#000000"}
halley_style = {label: "Florence", color: "#84B0B8"}
```

The static two-dimensional plot can be created using the following code:

```
# Generate a static 2D figure
frame2D = rame = plot_solar_system(
    epoch=now, outer=False
)
frame2D.plot(florence, **florence_style)
frame2D.plot(florence, **halley_style)
```

As a result, figure 4 is obtained.

The interactive three-dimensional plot can be created using the following code:

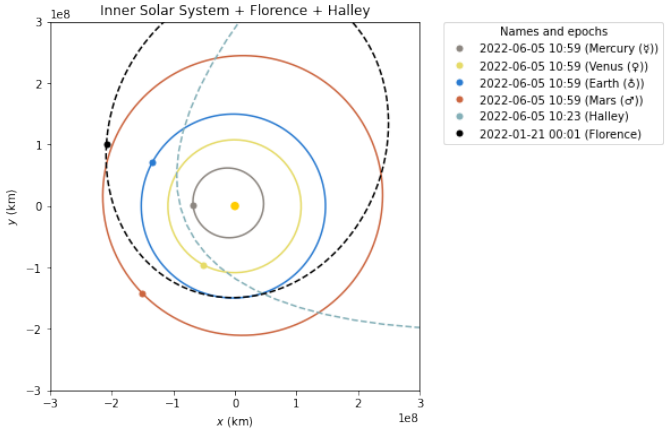


Fig. 4: *Two-dimensional view of the inner Solar System, Florence, and Halley.*

```
# Generate an interactive 3D figure
frame3D = rame = plot_solar_system(
    epoch=now, outer=False,
    use_3d=True, interactive=true
)
frame3D.plot(florence, **florence_style)
frame3D.plot(florence, **halley_style)
```

As a result, figure 5 is obtained.

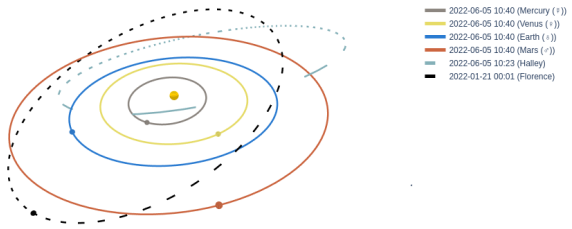


Fig. 5: *Three-dimensional view of the inner Solar System, Florence, and Halley.*

Commercial Earth satellites

Figure 6 gives a clear picture of the most important natural perturbations affecting satellites in LEO, namely: the first harmonic of the geopotential field J_2 (representing the attractor oblateness), the atmospheric drag, and the higher order harmonics of the geopotential field.

At least the most significant of these perturbations need to be taken into account when propagating LEO orbits, and therefore the methods for purely Keplerian motion are not enough. As seen above, poliastro implements a number of these perturbations already - however, numerical methods are much slower than analytical ones, and this can render them unsuitable for large scale simulations, satellite conjunction assesment, propagation in constrained hardware, and so forth.

To address this issue, semianalytical propagation methods were devised that attempt to strike a balance between the fast running times of analytical methods and the necessary inclusion of perturbation forces. One of such semianalytical methods are

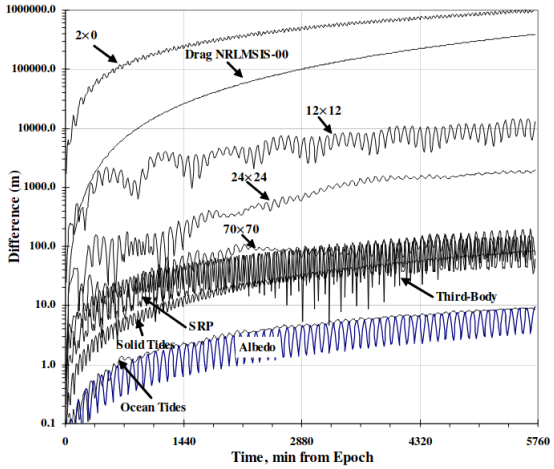


Fig. 6: *Natural perturbations affecting Low-Earth Orbit (LEO) motion (source: [VM07])*

the Simplified General Perturbation (SGP) models, first developed in [HK66] and then refined in [LC69] into what we know these days as the SGP4 propagator [HR80] [VCHK06]. Even though certain elements of the reference frame used by SGP4 are not properly specified [VCHK06] and that its accuracy might still be too limited for certain applications [Ko09] [Lar16], it is nowadays the most widely used propagation method thanks in large part to the dissemination of General Perturbations orbital data by the US 501(c)(3) CelesTrak (which itself obtains it from the 18th Space Defense Squadron of the US Space Force).

The starting point of SGP4 is a special element set that uses Brouwer mean orbital elements [Bro59] plus a ballistic coefficient based on an approximation of the atmospheric drag [LC69], and its results are expressed in a special coordinate system called True Equator Mean Equinox (TEME). Special care needs to be taken to avoid mixing mean elements with osculating elements, and to convert the output of the propagation to the appropriate reference frame. These element sets have been traditionally distributed in a compact text representation called Two-Line Element sets (TLEs) (see 7 for an example). However this format is quite cryptic and suffers from a number of shortcomings, so recently there has been a push to use the Orbit Data Messages international standard developed by the Consultive Committee for Space Data Systems (CCSDS 502.0-B-2).

```
1 25544U 98067A   22156.15037205  .00008547  00000+0  15823-3 0  9994
2 25544  51.6449  36.2070 0004577 196.3587 298.4146 15.49876730343319
```

Fig. 7: *Two-Line Element set (TLE) for the ISS (retrieved on 2022-06-05)*

At the moment, general perturbations data both in OMM and TLE format can be integrated with poliastro thanks to the `sgp4` Python library and the `Ephem` class as follows:

```
from astropy.coordinates import TEME, GCRS

from poliastro.ephem import Ephem
from poliastro.frames import Planes


def ephem_from_gp(sat, times):
    errors, rs, vs = sat.sgp4_array(times.jd1, times.jd2)
    if not (errors == 0).all():
        warn(
            "Some objects could not be propagated, "
```

```
            "proceeding with the rest",
            stacklevel=2,
        )
        rs = rs[errors == 0]
        vs = vs[errors == 0]
        times = times[errors == 0]

    cart_teme = CartesianRepresentation(
        rs << u.km,
        xyz_axis=-1,
        differentials=CartesianDifferential(
            vs << (u.km / u.s),
            xyz_axis=-1,
        ),
    )
    cart_gcrs = (
        TEME(cart_teme, obstime=times)
        .transform_to(GCRS(obstime=times))
        .cartesian
    )

    return Ephem(
        cart_gcrs,
        times,
        plane=Planes.EARTH_EQUATOR
    )
```

However, no native integration with SGP4 has been implemented yet in poliastro, for technical and non-technical reasons. On one hand, this propagator is too different from the other methods, and we have not yet devised how to add it to the library in a way that does not create confusion. On the other hand, adding such a propagator to poliastro would probably open the flood gates of corporate users of the library, and we would like to first devise a sustainability strategy for the project, which is addressed in the next section.

Future work

Despite the fact that poliastro has existed for almost a decade, for most of its history it has been developed by volunteers on their free time, and only in the past five years it has received funding through various Summer of Code programs (SOCIS 2017, GSOC 2018-2021) and institutional grants (NumFOCUS 2020, 2021). The funded work has had an overwhemingly positive impact on the project, however the lack of a dedicated maintainer has caused some technical debt to accrue over the years, and some parts of the project are in need of refactoring or better documentation.

Historically, poliastro has tried to implement algorithms that were applicable for all the planets in the Solar System, however some of them have proved to be very difficult to generalize for bodies other than the Earth. For cases like these, poliastro ships a `poliastro.earth` package, but going forward we would like to continue embracing a generic approach that can serve other bodies as well.

Several open source projects have successfully used poliastro or were created taking inspiration from it, like spacetech-ssa by IBM[1] or mubody [BBVPFSC22]. AGI (previously Analytical Graphics, Inc., now Ansys Government Initiatives) published a series of scripts to automate the commercial tool STK from Python leveraging poliastro[2]. However, we have observed that there is still lots of repeated code across similar open source libraries written in Python, which means that there is an opportunity to provide a "kernel" of algorithms that can be easily reused. Although `poliastro.core` started as a separate layer to isolate fast, non-safe functions as described above, we think we could move it to an external package so it can be depended upon by projects that do not want to use some of the higher level poliastro abstractions or drag its large number of heavy dependencies.

Finally, the sustainability of the project cannot yet be taken for granted: the project has reached a level of complexity that already warrants dedicated development effort that cannot be covered with short-lived grants. Such funding could potentially come from the private sector, but although there is evidence that several for-profit companies are using poliastro, we have very little information of how is it being used and what problems are those users having, let alone what avenues for funded work could potentially work. Organizations like the Libre Space Foundation advocate for a strong copyleft licensing model to convince commercial actors to contribute to the commons, but in principle that goes against the permissive licensing that the wider Scientific Python ecosystem, including poliastro, has adopted. With the advent of new business models and the ever increasing reliance in open source by the private sector, a variety of ways to engage commercial users and include them in the conversation exist. However, these have not been explored yet.

Acknowledgements

The authors would like to thank Prof. Michèle Lavagna for her original guidance and inspiration, David A. Vallado for his encouragement and for publishing the source code for the algorithms from his book for free, Dr. T.S. Kelso for his tireless efforts in maintaining CelesTrak, Alejandro Sáez for sharing the dream of a better way, Prof. Dr. Manuel Sanjurjo Rivo for believing in my work, Helge Eichhorn for his enthusiasm and decisive influence in poliastro, the whole OpenAstronomy collaboration for opening the door for us, the NumFOCUS organization for their immense support, and Alexandra Elbakyan for enabling scientific progress worldwide.

References

[AAA+76] United States Committee on Extension to the Standard Atmosphere, United States National Aeronautics, Space Administration, United States National Oceanic, Atmospheric Administration, and United States Air Force. *U.S. Standard Atmosphere, 1976*. NOAA - SIT 76-1562. National Oceanic and Amospheric [sic] Administration, 1976. URL: https://books.google.es/books?id=x488AAAAIAAJ.

[AAAA62] United States Committee on Extension to the Standard Atmosphere, United States National Aeronautics, Space Administration, and United States Environmental Science Services Administration. *U.S. Standard Atmosphere, 1962: ICAO Standard Atmosphere to 20 Kilometers; Proposed ICAO Extension to 32 Kilometers; Tables and Data to 700 Kilometers*. U.S. Government Printing Office, 1962. URL: https://books.google.es/books?id=fWdTAAAAMAAJ.

[Bat99] Richard H. Battin. *An Introduction to the Mathematics and Methods of Astrodynamics, Revised Edition*. American Institute of Aeronautics and Astronautics, Inc., Reston, VA, January 1999. URL: https://arc.aiaa.org/doi/book/10.2514/4.861543, doi:10.2514/4.861543.

[BBC+11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 13(2):31–39, March 2011. URL: http://ieeexplore.ieee.org/document/5582062/, doi:10.1109/MCSE.2010.118.

[BBVPFSC22] Juan Bermejo Ballesteros, José María Vergara Pérez, Alejandro Fernández Soler, and Javier Cubas. Mubody, an astrodynamics open-source Python library focused on libration points. Barcelona, Spain, April 2022. URL: https://sseasymposium.org/wp-content/uploads/2022/04/4thSSEA_AllAbstracts.pdf.

1. https://github.com/IBM/spacetech-ssa
2. https://github.com/AnalyticalGraphicsInc/STKCodeExamples/

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. URL: https://epubs.siam.org/doi/10.1137/141000671, doi:10.1137/141000671.

[Bro59] Dirk Brouwer. Solution of the problem of artificial satellite theory without drag. *The Astronomical Journal*, 64:378, November 1959. URL: http://adsabs.harvard.edu/cgi-bin/bib_query?1959AJ.....64..378B, doi:10.1086/107958.

[Bur67] E.G.C. Burt. On space manoeuvres with continuous thrust. *Planetary and Space Science*, 15(1):103–122, January 1967. URL: https://linkinghub.elsevier.com/retrieve/pii/0032063367900700, doi:10.1016/0032-0633(67)90070-0.

[CC10] Philip Herbert Cowell and Andrew Claude Crommelin. *Investigation of the Motion of Halley's Comet from 1759 to 1910*. Neill & Company, limited, 1910.

[Cha22] Kevin Charls. Recursive solution to Kepler's problem for elliptical orbits - application in robust Newton-Raphson and co-planar closest approach estimation. 2022. Publisher: Unpublished Version Number: 1. URL: https://rgdoi.net/10.13140/RG.2.2.18578.58563/1, doi:10.13140/RG.2.2.18578.58563/1.

[Con14] Bruce A. Conway. *Spacecraft trajectory optimization*. Number 29 in Cambridge aerospace series. Cambridge university press, Cambridge (GB), 2014.

[CR17] Juan Luis Cano Rodríguez. Study of analytical solutions for low-thrust trajectories. Master's thesis, Universidad Politécnica de Madrid, March 2017.

[DB83] J. M. A. Danby and T. M. Burkardt. The solution of Kepler's equation, I. *Celestial Mechanics*, 31(2):95–107, October 1983. URL: http://link.springer.com/10.1007/BF01686811, doi:10.1007/BF01686811.

[DCV21] Marilena Di Carlo and Massimiliano Vasile. Analytical solutions for low-thrust orbit transfers. *Celestial Mechanics and Dynamical Astronomy*, 133(7):33, July 2021. URL: https://link.springer.com/10.1007/s10569-021-10033-9, doi:10.1007/s10569-021-10033-9.

[Dro53] S. Drobot. On the foundations of Dimensional Analysis. *Studia Mathematica*, 14(1):84–99, 1953. URL: http://www.impan.pl/get/doi/10.4064/sm-14-1-84-99, doi:10.4064/sm-14-1-84-99.

[Dub73] G. N. Duboshin. Book Review: Samuel Herrick. Astrodynamics. *Soviet Astronomy*, 16:1064, June 1973. ADS Bibcode: 1973SvA....16.1064D. URL: https://ui.adsabs.harvard.edu/abs/1973SvA....16.1064D.

[Ede61] Theodore N. Edelbaum. Propulsion Requirements for Controllable Satellites. *ARS Journal*, 31(8):1079–1089, August 1961. URL: https://arc.aiaa.org/doi/10.2514/8.5723, doi:10.2514/8.5723.

[FCM13] Davide Farnocchia, Davide Bracali Cioci, and Andrea Milani. Robust resolution of Kepler's equation in all eccentricity regimes. *Celestial Mechanics and Dynamical Astronomy*, 116(1):21–34, May 2013. URL: http://link.springer.com/10.1007/s10569-013-9476-9, doi:10.1007/s10569-013-9476-9.

[Fin07] D Finkleman. "TLE or Not TLE?" That is the Question (AAS 07-126). *ADVANCES IN THE ASTRONAUTICAL SCIENCES*, 127(1):401, 2007. Publisher: Published for the American Astronautical Society by Univelt; 1999.

[GBP22] Mirko Gabelica, Ružica Bojčić, and Livia Puljak. Many researchers were not compliant with their published data sharing statement: mixed-methods study. *Journal of Clinical Epidemiology*, page S089543562200141X, May 2022. URL: https://linkinghub.elsevier.com/retrieve/pii/S089543562200141X, doi:10.1016/j.jclinepi.2022.05.019.

[Her71] Samuel Herrick. *Astrodynamics*. Van Nostrand Reinhold Co, London, New York, 1971.

[HK66] CG Hilton and JR Kuhlman. Mathematical models for the space defense center. *Philco-Ford Publication No. U-3871*, 17:28, 1966.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://www.nature.com/articles/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[HNW09] E. Hairer, S. P. Nørsett, and Gerhard Wanner. *Solving ordinary differential equations I: nonstiff problems*. Number 8 in Springer series in computational mathematics. Springer, Heidelberg ; London, 2nd rev. ed edition, 2009. OCLC: ocn620251790.

[HR80] Felix R. Hoots and Ronald L. Roehrich. Models for propagation of NORAD element sets. Technical report, Defense Technical Information Center, Fort Belvoir, VA, December 1980. URL: http://www.dtic.mil/docs/citations/ADA093554.

[Hun07] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. Publisher: IEEE COMPUTER SOC. doi:10.1109/MCSE.2007.55.

[IBD+20] Dario Izzo, Will Binns, Dariomm098, Alessio Mereta, Christopher Iliffe Sprague, Dhennes, Bert Van Den Abbeele, Chris Andre, Krzysztof Nowak, Nat Guy, Alberto Isaac Barquín Murguía, Pablo, Frédéric Chapoton, GiacomoAcciarini, Moritz V. Looz, Dietmarwo, Mike Heddes, Anatoli Babenia, Baptiste Fournier, Johannes Simon, Jonathan Willitts, Mateusz Polnik, Sanjeev Narayanaswamy, The Gitter Badger, and Jack Yarndley. esa/pykep: Optimize, October 2020. URL: https://zenodo.org/record/4091753, doi:10.5281/ZENODO.4091753.

[Inc15] Plotly Technologies Inc. Collaborative data science, 2015. Place: Montreal, QC Publisher: Plotly Technologies Inc. URL: https://plot.ly.

[Jac77] L. G. Jacchia. Thermospheric Temperature, Density, and Composition: New Models. *SAO Special Report*, 375, March 1977. ADS Bibcode: 1977SAOSR.375.....J. URL: https://ui.adsabs.harvard.edu/abs/1977SAOSR.375.....J.

[JGAZJT+18] Nathan J. Goldbaum, John A. ZuHone, Matthew J. Turk, Kacper Kowalik, and Anna L. Rosen. unyt: Handle, manipulate, and convert data with units in Python. *Journal of Open Source Software*, 3(28):809, August 2018. URL: http://joss.theoj.org/papers/10.21105/joss.00809, doi:10.21105/joss.00809.

[Kec97] Jean Albert Kechichian. Reformulation of Edelbaum's Low-Thrust Transfer Problem Using Optimal Control Theory. *Journal of Guidance, Control, and Dynamics*, 20(5):988–994, September 1997. URL: https://arc.aiaa.org/doi/10.2514/2.4145, doi:10.2514/2.4145.

[Ko09] TS Kelso and others. Analysis of the Iridium 33-Cosmos 2251 collision. *Advances in the Astronautical Sciences*, 135(2):1099–1112, 2009. Publisher: Citeseer.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, and others. *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, volume 2016. 2016.

[Lar16] Martin Lara. Analytical and Semianalytical Propagation of Space Orbits: The Role of Polar-Nodal Variables. In Gerard Gómez and Josep J. Masdemont, editors, *Astrodynamics Network AstroNet-II*, volume 44, pages 151–166. Springer International Publishing, Cham, 2016. Series Title: Astrophysics and Space Science Proceedings. URL: http://link.springer.com/10.1007/978-3-319-23986-6_11, doi:10.1007/978-3-319-23986-6_11.

[LC69] M. H. Lane and K. Cranford. An improved analytical drag theory for the artificial satellite problem. In *Astrodynamics Conference*, Princeton,NJ,U.S.A., August 1969. American Institute of Aeronautics and Astronautics. URL: https://arc.aiaa.org/doi/10.2514/6.1969-925, doi:10.2514/6.1969-925.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, pages 1–6, Austin, Texas, 2015. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2833157.2833162, doi:10.1145/2833157.2833162.

[Mar95] F. Landis Markley. Kepler Equation solver. *Celestial Mechanics & Dynamical Astronomy*, 63(1):101–111,

1995. URL: http://link.springer.com/10.1007/BF00691917, doi:10.1007/BF00691917.

[Mik87] Seppo Mikkola. A cubic approximation for Kepler's equation. *Celestial Mechanics*, 40(3-4):329–334, September 1987. URL: http://link.springer.com/10.1007/BF01235850, doi:10.1007/BF01235850.

[MKDVB+19] Michael Mommert, Michael Kelley, Miguel De Val-Borro, Jian-Yang Li, Giannina Guzman, Brigitta Sipőcz, Josef Ďurech, Mikael Granvik, Will Grundy, Nick Moskovitz, Antti Penttilä, and Nalin Samarasinha. sbpy: A Python module for small-body planetary astronomy. *Journal of Open Source Software*, 4(38):1426, June 2019. URL: http://joss.theoj.org/papers/10.21105/joss.01426, doi:10.21105/joss.01426.

[noa] Astrodynamics.jl. URL: https://github.com/JuliaSpace/Astrodynamics.jl.

[noa18] gpredict, January 2018. URL: https://github.com/csete/gpredict/releases/tag/v2.2.1.

[noa20] GMAT, July 2020. URL: https://sourceforge.net/projects/gmat/files/GMAT/GMAT-R2020a/.

[noa21a] nyx, November 2021. URL: https://gitlab.com/nyx-space/nyx/-/tags/1.0.0.

[noa21b] SatNOGS, October 2021. URL: https://gitlab.com/librespacefoundation/satnogs/satnogs-client/-/tags/1.7.

[noa22a] beyond, January 2022. URL: https://pypi.org/project/beyond/0.7.4/.

[noa22b] celestlab, January 2022. URL: https://atoms.scilab.org/toolboxes/celestlab/3.4.1.

[noa22c] Orekit, June 2022. URL: https://gitlab.orekit.org/orekit/orekit/-/releases/11.2.

[noa22d] SPICE, January 2022. URL: https://naif.jpl.nasa.gov/naif/toolkit.html.

[noa22e] tudatpy, January 2022. URL: https://github.com/tudat-team/tudatpy/releases/tag/0.6.0.

[OG86] A. W. Odell and R. H. Gooding. Procedures for solving Kepler's equation. *Celestial Mechanics*, 38(4):307–334, April 1986. URL: http://link.springer.com/10.1007/BF01238923, doi:10.1007/BF01238923.

[Pol97] James E Pollard. Simplified approach for assessment of low-thrust elliptical orbit transfers. In *25th International Electric Propulsion Conference, Cleveland, OH*, pages 97–160, 1997.

[Pol98] James Pollard. Evaluation of low-thrust orbital maneuvers. In *34th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Cleveland,OH,U.S.A., July 1998. American Institute of Aeronautics and Astronautics. URL: https://arc.aiaa.org/doi/10.2514/6.1998-3486, doi:10.2514/6.1998-3486.

[Pol00] J. E. Pollard. Simplified analysis of low-thrust orbital maneuvers. Technical report, Defense Technical Information Center, Fort Belvoir, VA, August 2000. URL: http://www.dtic.mil/docs/citations/ADA384536.

[PP13] Adonis Reinier Pimienta-Penalver. *Accurate Kepler equation solver without transcendental function evaluations*. State University of New York at Buffalo, 2013.

[Rho20] Brandon Rhodes. Skyfield: Generate high precision research-grade positions for stars, planets, moons, and Earth satellites, February 2020.

[SSM18] Victoria Stodden, Jennifer Seiler, and Zhaokun Ma. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, 115(11):2584–2589, March 2018. URL: https://pnas.org/doi/full/10.1073/pnas.1708290115, doi:10.1073/pnas.1708290115.

[TPWS+18] The Astropy Collaboration, A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, A. Ginsburg, J. T. VanderPlas, L. D. Bradley, D. Pérez-Suárez, M. de Val-Borro, (Primary Paper Contributors), T. L. Aldcroft, K. L. Cruz, T. P. Robitaille, E. J. Tollerud, (Astropy Coordination Committee), C. Ardelean, T. Babej, Y. P. Bach, M. Bachetti, A. V. Bakanov, S. P. Bamford, G. Barentsen, P. Barmby, A. Baumbach, K. L. Berry, F. Biscani, M. Boquien, K. A. Bostroem, L. G. Bouma, G. B. Brammer, E. M. Bray, H. Breytenbach, H. Buddelmeijer, D. J. Burke, G. Calderone, J. L. Cano Rodríguez, M. Cara, J. V. M. Cardoso, S. Cheedella, Y. Copin, L. Corrales, D. Crichton, D. D'Avella, C. Deil, É. Depagne, J. P. Dietrich, A. Donath, M. Droettboom, N. Earl, T. Erben, S. Fabbro, L. A. Ferreira, T. Finethy, R. T. Fox, L. H. Garrison, S. L. J. Gibbons, D. A. Goldstein, R. Gommers, J. P. Greco, P. Greenfield, A. M. Groener, F. Grollier, A. Hagen, P. Hirst, D. Homeier, A. J. Horton, G. Hosseinzadeh, L. Hu, J. S. Hunkeler, Ž. Ivezić, A. Jain, T. Jenness, G. Kanarek, S. Kendrew, N. S. Kern, W. E. Kerzendorf, A. Khvalko, J. King, D. Kirkby, A. M. Kulkarni, A. Kumar, A. Lee, D. Lenz, S. P. Littlefair, Z. Ma, D. M. Macleod, M. Mastropietro, C. McCully, S. Montagnac, B. M. Morris, M. Mueller, S. J. Mumford, D. Muna, N. A. Murphy, S. Nelson, G. H. Nguyen, J. P. Ninan, M. Nöthe, S. Ogaz, S. Oh, J. K. Parejko, N. Parley, S. Pascual, R. Patil, A. A. Patil, A. L. Plunkett, J. X. Prochaska, T. Rastogi, V. Reddy Janga, J. Sabater, P. Sakurikar, M. Seifert, L. E. Sherbert, H. Sherwood-Taylor, A. Y. Shih, J. Sick, M. T. Silbiger, S. Singanamalla, L. P. Singer, P. H. Sladen, K. A. Sooley, S. Sornarajah, O. Streicher, P. Teuben, S. W. Thomas, G. R. Tremblay, J. E. H. Turner, V. Terrón, M. H. van Kerkwijk, A. de la Vega, L. L. Watkins, B. A. Weaver, J. B. Whitmore, J. Woillez, V. Zabalza, and (Astropy Contributors). The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *The Astronomical Journal*, 156(3):123, August 2018. URL: https://iopscience.iop.org/article/10.3847/1538-3881/aabc4f, doi:10.3847/1538-3881/aabc4f.

[VCHK06] David Vallado, Paul Crawford, Ricahrd Hujsak, and T.S. Kelso. Revisiting Spacetrack Report #3. In *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, Keystone, Colorado, August 2006. American Institute of Aeronautics and Astronautics. URL: https://arc.aiaa.org/doi/10.2514/6.2006-6753, doi:10.2514/6.2006-6753.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, SciPy 1.0 Contributors, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert-Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vinícius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, and Yoshiki Vázquez-Baeza. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, March 2020. URL: http://www.nature.com/articles/s41592-019-0686-2, doi:10.1038/s41592-019-0686-2.

[VM07] David A. Vallado and Wayne D. McClain. *Fundamentals of astrodynamics and applications*. Number 21 in Space technology library. Microcosm Press [u.a.], Hawthorne, Calif., 3. ed., 1. printing edition, 2007.

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

1001745, doi:10.1371/journal.pbio.1001745.

[WIO85] M. J. H. Walker, B. Ireland, and Joyce Owens. A set modified equinoctial orbit elements. *Celestial Mechanics*, 36(4):409–419, August 1985. URL: http://link.springer.com/10.1007/BF01227493, doi:10.1007/BF01227493.

A New Python API for Webots Robotics Simulations

Justin C. Fisher[‡*]

Abstract—Webots is a popular open-source package for 3D robotics simulations. It can also be used as a 3D interactive environment for other physics-based modeling, virtual reality, teaching or games. Webots has provided a simple API allowing Python programs to control robots and/or the simulated world, but this API is inefficient and does not provide many "pythonic" conveniences. A new Python API for Webots is presented that is more efficient and provides a more intuitive, easily usable, and "pythonic" interface.

Index Terms—Webots, Python, Robotics, Robot Operating System (ROS), Open Dynamics Engine (ODE), 3D Physics Simulation

1. Introduction

Webots is a popular open-source package for 3D robotics simulations [Mic01], [Webots]. It can also be used as a 3D interactive environment for other physics-based modeling, virtual reality, teaching or games. Webots uses the Open Dynamics Engine [ODE], which allows physical simulations of Newtonian bodies, collisions, joints, springs, friction, and fluid dynamics. Webots provides the means to simulate a wide variety of robot components, including motors, actuators, wheels, treads, grippers, light sensors, ultrasound sensors, pressure sensors, range finders, radar, lidar, and cameras (with many of these sensors drawing their inputs from GPU processing of the simulation). A typical simulation will involve one or more robots, each with somewhere between 3 and 30 moving parts (though more would be possible), each running its own controller program to process information taken in by its sensors to determine what control signals to send to its devices. A simulated world typically involves a ground surface (which may be a sloping polygon mesh) and dozens of walls, obstacles, and/or other objects, which may be stationary or moving in the physics simulation.

Webots has historically provided a simple Python API, allowing Python programs to control individual robots or the simulated world. This Python API is a thin wrapper over a C++ API, which itself is a wrapper over Webots' core C API. These nested layers of API-wrapping are inefficient. Furthermore, this API is not very "pythonic" and did not provide many of the conveniences that help to make development in Python be fast, intuitive, and easy to learn. This paper presents a new Python API [NewAPI01] that more efficiently interfaces directly with the Webots C API and provides a more intuitive, easily usable, and "pythonic" interface for controlling Webots robots and simulations.

* *Corresponding author: fisher@smu.edu*
‡ *Southern Methodist University, Department of Philosophy*

In qualitative terms, the old API feels like one is awkwardly using Python to call C and C++ functions, whereas the new API feels much simpler, much easier, and like it is fully intended for Python. Here is a representative (but far from comprehensive) list of examples:

- Unlike the old API, the new API contains helpful Python type annotations and docstrings.
- Webots employs many vectors, e.g., for 3D positions, 4D rotations, and RGB colors. The old API typically treats these as lists or integers (24-bit colors). In the new API these are Vector objects, with conveniently addressable components (e.g. `vector.x` or `color.red`), convenient helper methods like `vector.magnitude` and `vector.unit_vector`, and overloaded vector arithmetic operations, akin to (and interoperable with) NumPy arrays.
- The new API also provides easy interfacing between high-resolution Webots sensors (like cameras and Lidar) and Numpy arrays, to make it much more convenient to use Webots with popular Python packages like Numpy [NumPy], [Har01], Scipy [Scipy], [Vir01], PIL/PILLOW [PIL] or OpenCV [OpenCV], [Brad01]. For example, converting a Webots camera image to a NumPy array is now as simple as `camera.array` and this now allows the array to share memory with the camera, making this extremely fast regardless of image size.
- The old API often requires that all function parameters be given explicitly in every call, whereas the new API gives many parameters commonly used default values, allowing them often to be omitted, and keyword arguments to be used where needed.
- Most attributes are now accessible (and alterable, when applicable) by pythonic properties like `motor.velocity`.
- Many devices now have Python methods like `__bool__` overloaded in intuitive ways. E.g., you can now use `if bumper` to detect if a bumper has been pressed, rather than the old `if bumper.getValue()`.
- Pythonic container-like interfaces are now provided. You may now use `for target in radar` to iterate through the various targets a radar device has detected or `for packet in receiver` to iterate through communication packets that a receiver device has received (and it now automatically handles a wide variety of Python objects, not just strings).
- The old API requires supervisor controllers to use a wide variety of separate functions to traverse and interact with the simulation's scene tree, including dif-

ferent functions for different VRML datatypes (like `SFVec3f` or `MFInt32`). The new API automatically handles these datatypes and translates intuitive Python syntax (like dot-notation and square-bracket indexing) to the Webots equivalents. E.g., you can now move a particular crate 1 meter in the x direction using a command like `world.CRATES[3].translation += [1,0,0]`. Under the old API, this would require numerous function calls (calling `getNodeFromDef` to find the CRATES node, `getMFNode` to find the child with index 3, `getSFField` to find its translation field, and `getSFVec3f` to retrieve that field's value, then some list manipulation to alter the x-component of that value, and finally a call to `setSFVec3f` to set the new value).

As another example illustrating how much easier the new API is to use, here are two lines from Webots' sample `supervisor_draw_trail`, as it would appear in the old Python API.

```
f = supervisor.getField(supervisor.getRoot(),
                        "children")
f.importMFNodeFromString(-1, trail_plan)
```

And here is how that looks written using the new API:

```
world.children.append(trail_plan)
```

The new API is mostly backwards-compatible with the old Python Webots API, and provides an option to display deprecation warnings with helpful advice for changing to the new API.

The new Python API is planned for inclusion in an upcoming Webots release, to replace the old one. In the meantime, an early-access version is available, distributed under Apache 2.0 licence, the same permissibe open-source license that Webots is distributed under.

In what follows, the history and motivation for this new API is discussed, including its use in teaching an interdisciplinary undergraduate Cognitive Science course called Minds, Brains and Robotics. Some of the design decisions for the new API are discussed, which will not only aid in understanding it, but also have broader relevance to parallel dilemmas that face many other software developers. And some metrics are given to quantify how the new API has improved over the old.

2. History and Motivation.

Much of this new API was developed by the author in the course of teaching an interdisciplinary Southern Methodist University undergraduate Cognitive Science course entitled Minds, Brains and Robotics (PHIL 3316). Before the Covid pandemic, this course had involved lab activities where students build and program physical robots. The pandemic forced these activities to become virtual. Fortunately, Webots simulations actually have many advantages over physical robots, including not requiring any specialized hardware (beyond a decent personal computer), making much more interesting uses of altitude rather than having the robots confined to a safely flat surface, allowing robots to engage in dangerous or destructive activities that would be risky or expensive with physical hardware, allowing a much broader array of sensors including high-resolution cameras, and enabling full-fledged neural network and computational vision simulations. For example, an early activity in this class involves building Braitenburg-style vehicles [Bra01] that use light sensors and cameras to detect a lamp carried by a hovering drone, as well as ultrasound and touch sensors to detect obstables. Using these sensors, the robots navigate towards the lamp in a cluttered playground sandbox that includes sloping sand, an exterior wall, and various obstacles including a puddle of water and platforms from which robots may fall.

This interdisciplinary class draws students with diverse backgrounds, and programming skills. Accomodating those with fewer skills required simplifying many of the complexities of the old Webots API. It also required setting up tools to use Webots "supervisor" powers to help manipulate the simulated world, e.g. to provide students easier customization options for their robots. The old Webots API makes the use of such supervisor powers tedious and difficult, even for experienced coders, so this practically required developing new tools to streamline the process. These factors led to the development of an interface that would be much easier for novice students to adapt to, and that would make it much easier for an experienced programmer to make much use of supervisor powers to manipulate the simulated world. Discussion of this with the core Webots development team then led to the decision to incorporate these improvements into Webots, where they can be of benefit to a much broader community.

3. Design Decisions.

This section discusses some design decisions that arose in developing this API, and discusses the factors that drove these decisions. This may help give the reader a better understanding of this API, and also of relevant considerations that would arise in many other development scenarios.

3.1. Shifting from functions to properties.

The old Python API for Webots consists largely of methods like `motor.getVelocity()` and `motor.setVelocity(new_velocity)`. In the new API these have quite uniformly been changed to Python properties, so these purposes are now accomplished with `motor.velocity` and `motor.velocity = new_velocity`.

Reduction of wordiness and punctuation helps to make programs easier to read and to understand, and it reduces the cognitive load on coders. However, there are also drawbacks.

One drawback is that properties can give the mistaken impression that some attributes are computationally cheap to get or set. In cases where this impression would be misleading, more traditional method calls were retained and/or the comparative expense of the operation was clearly documented.

Two other drawbacks are related. One is that inviting ordinary users to assign properties to API objects might lead them to assign other attributes that could cause problems. Since Python lacks true privacy protections, it has always faced this sort of worry, but this worry becomes even worse when users start to feel familiar moving beyond just using defined methods to interact with an object.

Relatedly, Python debugging provides direct feedback in cases where a user misspells `motor.setFoo(v)` but not when someone mispells 'motor.foo = v'. If a user inadvertently types `motor.setFool(v)` they will get an `AttributeError` noting that `motor` lacks a `setFool` attribute. But if a user inadvertently types `motor.fool = v`, then Python will silently create a new `.fool` attribute for `motor` and the user will often have no idea what has gone wrong.

These two drawbacks both involve users setting an attribute they shouldn't: either an attribute that has another purpose, or one

that doesn't. Defenses against the first include "hiding" important attributes behind a leading "_", or protecting them with a Python property, which can also help provide useful doc-strings. Unfortunately it's much harder to protect against misspellings in this piece-meal fashion.

This led to the decision to have robot devices like motors and cameras employ a blanket `__setattr__` that will generate warnings if non-property attributes of devices are set from outside the module. So the user who inadvertently types `motor.fool = v` will immediately be warned of their mistake. This does incur a performance cost, but that cost is often worthwhile when it saves development time and frustration. For cases when performance is crucial, and/or a user wants to live dangerously and meddle inside API objects, this layer of protection can be deactivated.

An alternative approach, suggested by Matthew Feickert, would have been to use `__slots__` rather than an ordinary `__dict__` to store device attributes, which would also have the effect of raising an error if users attempt to modify unexpected attributes. Not having a `__dict__` can make it harder to do some things like cached properties and multiple inheritance. But in cases where such issues don't arise or can be worked around, readers facing similar challenges may find `__slots__` to be a preferable solution.

3.2 Backwards Compatibility.

The new API offers many new ways of doing things, many of which would seem "better" by most metrics, with the main drawback being just that they differ from old ways. The possibility of making a clean break from the old API was considered, but that would stop old code from working, alienate veteran users, and risk causing a schism akin to the deep one that arose between Python 2 and Python 3 communities when Python 3 opted against backwards compatibility.

Another option would have been to refrain from adding a "new-and-better" feature to avoid introducing redundancies or backward incompatibilities. But that has obvious drawbacks too.

Instead, a compromise was typically adopted: to provide both the "new-and-better" way and the "worse-old" way. This redundancy was eased by shifting from `getFoo` / `setFoo` methods to properties, and from `CamelCase` to pythonic `snake_case`, which reduced the number of name collisions between old and new. Employing the "worse-old" way leads to a deprecation warning that includes helpful advice regarding shifting to the "new-and-better" way of doing things. This may help users to transition more gradually to the new ways, or they can shut these warnings off to help preserve good will, and hopefully avoid a schism.

3.3 Separating `robot` and `world`.

In Webots there is a distinction between "ordinary robots" whose capabilities are generally limited to using the robot's own devices, and "supervisor robots" who share those capabilities, but also have virtual omniscience and omnipotence over most aspects of the simulated world. In the old API, supervisor controller programs import a `Supervisor` subclass of `Robot`, but typically still call this unusually powerful robot `robot`, which has led to many confusions.

In the new API these two sorts of powers are strictly separated. Importing `robot` provides an object that can be used to control the devices in the robot itself. Importing `world` provides an object that can be used to observe and enact changes anywhere in the simulated world (presuming that the controller has such permissions, of course). In many use cases, supervisor robots don't actually have bodies and devices of their own, and just use their supervisor powers incorporeally, so all they will need is `world`. In the case where a robot's controller wants to exert both forms of control, it can import both `robot` to control its own body, and `world` to control the rest of the world.

This distinction helps to make things more intuitively clear. It also frees `world` from having all the properties and methods that `robot` has, which in turn reduces the risk of name-collisions as `world` takes on the role of serving as the root of the proxy scene tree. In the new API, `world.children` refers to the `children` field of the root of the scene tree which contains (almost) all of the simulated world, `world.WorldInfo` refers to one of these children, a `WorldInfo` node, and `world.ROBOT2` dynamically returns a node within the world whose Webots DEF-name is "ROBOT2". These uses of `world` would have been much less intuitive if users thought of `world` as being a special sort of robot, rather than as being their handle on controlling the simulated world. Other sorts of supervisor functionality also are very intuitively associated with `world`, like `world.save(filename)` to save the state of the simulated world, or `world.mode = 'PAUSE'`.

Having `world.attributes` dynamically fetch nodes and fields from the scene tree did come with some drawbacks. There is a risk of name-collisions, though these are rare since Webots field-names are known in advance, and nodes are typically sought by ALL-CAPS DEF-names, which won't collide with `world` 's lower-case and MixedCase attributes. Linters like MyPy and PyCharm also cannot anticipate such dynamic references, which is unfortunate, but does not stop such dynamic references from being extremely useful.

4. Readability Metrics

A main advantage of the new API is that it allows Webots controllers to be written in a manner that is easier for coders to read, write, and understand. Qualitatively, this difference becomes quite apparent upon a cursory inspection of examples like the one given in section 1. As another representative example, here are three lines from Webots' included `supervisor_draw_trail` sample as they would appear in the old Python API:

```
trail_node = world.getFromDef("TRAIL")
point_field = trail_node.getField("coord")\
                        .getSFNode()\
                        .getField("point")
index_field = trail_node.getField("coordIndex")
```

And here is their equivalent in the new API:

```
point_field = world.TRAIL.coord.point
index_field = world.TRAIL.coordIndex
```

Brief inspection should reveal that the latter code is much easier to read, write and understand, not just because it is shorter, but also because its punctuation is limited to standard Python syntax for traversing attributes of objects, because it reduces the need to introduce new variables like `trail_node` for things that it already makes easy to reference (via `world.TRAIL`, which the new API automatically caches for fast repeat reference), and because it invisibly handles selecting appropriate C-API functions like `getField` and `getSFNode`, saving the user from needing to learn and remember all these functions (of which there are many).

Metric	New API	Old API
Lines of Code (with blanks, comments)	43	49
Source Lines of Code (without those)	29	35
Logical Lines of Code (single commands)	27	38
Cyclomatic Complexity	5 (A)	8 (B)

TABLE 1
Length and Complexity Metrics. Raw measures for `supervisor_draw_trail` as it would be written with the new Python API for Webots or the old Python API for Webots. The "lines of codes" measures differ with respect to how they count blank lines, comments, and lines that combine multiple commands. Cyclomatic complexity measures the number of potential branching points in the code.

Halstead Metric	New API	Old API
Vocabulary = (n1)operators+(n2)operands	18	54
Length = (N1)operator + (N2)operand instances	38	99
Volume = Length * $\log_2$(Vocabulary)	158	570
Difficulty = (n1 * N2) / (2 * n2)	4.62	4.77
Effort = Difficulty * Volume	731	2715
Time = Effort / 18	41	151
Bugs = Volume / 3000	0.05	0.19

TABLE 2
Halstead Metrics. Halstead metrics for `supervisor_draw_trail` as it would be written with the new and old Python API's for Webots. Lower numbers are commonly construed as being better.

This intuitive impression is confirmed by automated metrics for code readability. The measures in what follows consider the full `supervisor_draw_trail` sample controller (from which the above snippet was drawn), since this is the Webots sample controller that makes the most sustained use of supervisor functionality to perform a fairly plausible supervisor task (maintaining the position of a streamer that trails behind the robot). Webots provides this sample controller in C [SDTC], but it was re-implemented using both the Old Python API and the New Python API [Metrics], maintaining straightforward correspondence between the two, with the only differences being directly due to the differences in the API's.

Some raw measures for the two controllers are shown in Table 1. These were gathered using the Radon code-analysis tools [Radon]. (These metrics, as well as those below, may be reproduced by (1) installing Radon [Radon], (2) downloading the source files to compare and the script for computing Metrics [Metrics], (3) ensuring that the path at the top of the script refers to the local location of the source files to be compared, and (4) running this script.) Multiple metrics are reported because theorists disagree about which are most relevant in assessing code readability, because some of these play a role in computing other metrics discussed below, and because this may help to allay potential worries that a few favorable metrics might have been cherry-picked. This paper provides some explanation of these metrics and of their potential significance, while remaining neutral regarding which, if any, of these metrics is best.

The "lines of code" measures reflect that the new API makes it easier to do more things with less code. The measures differ in how they count blank lines, comments, multi-line statements, and multi-statement lines like `if p: q()`. Line counts can be misleading, especially when the code with fewer lines has longer lines, though upcoming measures will show that that is not the case here.

Cyclomatic Complexity counts the number of potential branching points that appear within the code, like `if`, `while` and `for`. [McC01] Cyclomatic Complexity is strongly correlated with other plausible measures of code readability involving indentation structure [Hin01]. The new API's score is lower/"better" due to its automatically converting vector-like values to the format needed for importing new nodes into the Webots simulation, and due to its automatic caching allowing a simpler loop to remove unwanted nodes. By Radon's reckoning this difference in complexity already gives the old API a "B" grade, as compared to the new API's "A". These complexity measures would surely rise in more complex controllers employed in larger simulations, but they would rise less quickly under the new API, since it provides many simpler ways of doing things, and need never do any worse since it provides backwards-compatible options.

Another collection of classic measures of code readability was developed by Halstead. [Hal01] These measures (especially volume) have been shown to correlate with human assessments of code readability [Bus01], [Pos01]. These measures generally penalize a program for using a "vocabulary" involving more operators and operands. Table 2 shows these metrics, as computed by Radon. (Again all measures are reported, while remaining neutral about which are most significant.) The new API scores significantly lower/"better" on these metrics, due in large part to its automatically selecting among many different C-API calls without these needing to appear in the user's code. E.g. having `motor.velocity` as a unified property involves fewer unique names than having users write both `setVelocity()` and `getVelocity()`, and often forming a third local `velocity` variable. And having `world.children[-1]` access the last child that field in the simulation saves having to count `getField`, and `getMFNode` in the vocabulary, and often also saves forming additional local variables for nodes or fields gotten in this way. Both of these factors also help the new API to greatly reduce parentheses counts.

Lastly, the Maintainability Index and variants thereof are intended to measure of how easy to support and change source code is. [Oman01] Variants of the Maintainability Index are commonly used, including in Microsoft Visual Studio. These measures combine Halstead Volume, Source Lines of Code, and Cyclomatic Complexity, all mentioned above, and two variants (SEI and Radon) also provide credit for percentage of comment lines. (Both samples compared here include 5 comment lines, but these compose a higher percentage of the new API's shorter code). Different versions of this measure weight and curve these factors somewhat differently, but since the new API outperforms the old on each factor, all versions agree that it gets the higher/"better" score, as shown in Table 3. (These measures were computed based on the input components as counted by Radon.)

There are potential concerns about each of these measures of code readability, and one can easily imagine playing a form of "code golf" to optimize some of these scores without actually improving readability (though it would be difficult to do this for all scores at once). Fortunately, most plausible measures of readability have been observed to be strongly correllated across ordinary cases, [Pos01] so the clear and unanimous agreement between these measures is a strong confirmation that the new API is indeed

Maintainability Index version	New API	Old API
Original [Oman01]	89	79
Software Engineering Institute	78	62
Microsoft Visual Studio	52	46
Radon	82	75

TABLE 3
Maintainability Index Metrics. Maintainability Index metrics for `supervisor_draw_trail` as it would be written with the new and old versions of the Python API for Webots, according to different versions of the Maintainability Index. Higher numbers are commonly construed as being better.

more readable. Other plausible measures of readability would take into account factors like whether the operands are ordinary English words, [Sca01] or how deeply nested (or indented) the code ends up being, [Hin01] both of which would also favor the new API. So the mathematics confirm what was likely obvious from visual comparison of code samples above, that the new API is indeed more "readable" than the old.

5. Conclusions

A new Python API for Webots robotic simulations was presented. It more efficiently interfaces directly with the Webots C API and provides a more intuitive, easily usable, and "pythonic" interface for controlling Webots robots and simulations. Motivations for the API and some of its design decisions were discussed, including decisions use python properties, to add new functionality alongside deprecated backwards compatibility, and to separate robot and supervisor/world functionality. Advantages of the new API were discussed and quantified using automated code readability metrics.

More Information

An early-access version of the new API and a variety of sample programs and metric computations: https://github.com/Justin-Fisher/new_python_api_for_webots

Lengthy discussion of the new API and its planned inclusion in Webots: https://github.com/cyberbotics/webots/pull/3801

Webots home page, including free download of Webots: https://cyberbotics.com/

REFERENCES

[Brad01] Bradski, G. The OpenCV Library. Dr Dobb's Journal of Software Tools. 2000.

[Bra01] Braitenberg, V. *Vehicles: Experiments in synthetic psychology.* Cambridge, MA: MIT Press. 1984.

[Bus01] Buse, R and W Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4): 546-58. 2010. doi: 10.1109/TSE.2009.70.

[Metrics] Fisher, J. Readability Metrics for a New Python API for Webots Robotics Simulations. 2022. doi: 10.5281/zenodo.6813819.

[Hal01] Halstead, M. *Elements of software science.* Elsevier New York. 1977.

[Har01] Harris, C., K. Millman, S. van der Walt, et al. Array programming with NumPy. *Nature* 585, 357–62. 2020. doi: 10.1038/s41586-020-2649-2.

[Hin01] Hindle, A, MW Godfrey and RC Holt. "Reading beside the lines: Indentation as a proxy for complexity metric." Program Comprehension. The 16th IEEE International Conference, 133-42. 2008. doi: 10.1109/icpc.2008.13.

[McC01] McCabe, TJ. "A Complexity Measure" , 2(4): 308-320. 1976.

[Mic01] Michel, O. "Webots: Professional Mobile Robot Simulation. *Journal of Advanced Robotics Systems.* 1(1): 39-42. 2004. doi: 10.5772/5618.

[NewAPI01] https://github.com/Justin-Fisher/new_python_api_for_webots

[NumPy] Numerical Python (NumPy). https://www.numpy.org

[ODE] Open Dynamics Engine. https://www.ode.org/

[Oman01] Oman, P and J Hagemeister. "Metrics for assessing a software system's maintainability," *Proceedings Conference on Software Maintenance*, 337-44. 1992. doi: 10.1109/ICSM.1992.242525.

[OpenCV] Open Source Computer Vision Library for Python. https://github.com/opencv/opencv-python

[PIL] Python Imaging Library. https://python-pillow.org/

[Pos01] Posnet, D, A Hindle and P Devanbu. "A simpler model of software readability." *Proceedings of the 8th working conference on mining software repositories*, 73-82. 2011.

[Radon] Radon. https://radon.readthedocs.io/en/latest/index.html

[Sca01] Scalabrino, S, M Linares-Vasquez, R Oliveto and D Poshyvanyk. "A Comprehensive Model for Code Readability." *Jounal of Software: Evolution and Process*, 1-29. 2017. doi: 10.1002/smr.1958.

[Scipy] https://www.scipy.org

[SDTC] https://cyberbotics.com/doc/guide/samples-howto#supervisor_draw_trail-wbt

[SDTNew] https://github.com/Justin-Fisher/new_python_api_for_webots/blob/d180bcc7f505f8168246bee379f8067dfaf373ea/webots_new_python_api_samples/controllers/supervisor_draw_trail_python/supervisor_draw_trail_new_api_bare_bones.py

[SDTOld] https://github.com/Justin-Fisher/new_python_api_for_webots/blob/d180bcc7f505f8168246bee379f8067dfaf373ea/webots_new_python_api_samples/controllers/supervisor_draw_trail_python/supervisor_draw_trail_old_api_bare_bones.py

[Vir01] Virtanen, P, R. Gommers, T. Oliphant, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17(3), 261-72. 2020. doi: 10.1038/s41592-019-0686-2.

[Webots] Webots Open Source Robotic Simulator. https://cyberbotics.com/

pyAudioProcessing: Audio Processing, Feature Extraction, and Machine Learning Modeling

Jyotika Singh[‡*]

Abstract—pyAudioProcessing is a Python based library for processing audio data, constructing and extracting numerical features from audio, building and testing machine learning models, and classifying data with existing pre-trained audio classification models or custom user-built models. MATLAB is a popular language of choice for a vast amount of research in the audio and speech processing domain. On the contrary, Python remains the language of choice for a vast majority of machine learning research and functionality. This library contains features built in Python that were originally published in MATLAB. pyAudioProcessing allows the user to compute various features from audio files including Gammatone Frequency Cepstral Coefficients (GFCC), Mel Frequency Cepstral Coefficients (MFCC), spectral features, chroma features, and others such as beat-based and cepstrum-based features from audio. One can use these features along with one's own classification backend or any of the popular scikit-learn classifiers that have been integrated into pyAudioProcessing. Cleaning functions to strip unwanted portions from the audio are another offering of the library. It further contains integrations with other audio functionalities such as frequency and time-series visualizations and audio format conversions. This software aims to provide machine learning engineers, data scientists, researchers, and students with a set of baseline models to classify audio. The library is available at https://github.com/jsingh811/pyAudioProcessing and is under GPL-3.0 license.

Index Terms—pyAudioProcessing, audio processing, audio data, audio classification, audio feature extraction, gfcc, mfcc, spectral features, spectrogram, chroma

Introduction

The motivation behind this software is to make available complex audio features in Python for a variety of audio processing tasks. Python is a popular choice for machine learning tasks. Having solutions for computing complex audio features using Python enables easier and unified usage of Python for building machine learning algorithms on audio. This not only implies the need for resources to guide solutions for audio processing, but also signifies the need for Python guides and implementations to solve audio and speech cleaning, transformation, and classification tasks.

Different data processing techniques work well for different types of data. For example, in natural language processing, word embedding is a term used for the representation of words for text analysis, typically in the form of a real-valued numerical vector that encodes the meaning of the word such that the words that are closer in the vector space are expected to be similar in meaning [Wik22b]. Word embeddings work great for many applications surrounding textual data [JS21]. However, passing numbers, an audio signal, or an image through a word embeddings generation method is not likely to return any meaningful numerical representation that can be used to train machine learning models. Different data types correlate with feature formation techniques specific to their domain rather than a one-size-fits-all. These methods for audio signals are very specific to audio and speech signal processing, which is a domain of digital signal processing. Digital signal processing is a field of its own and is not feasible to master in an ad-hoc fashion. This calls for the need to have sought-after and useful processes for audio signals to be in a ready-to-use state by users.

There are two popular approaches for feature building in audio classification tasks.

1. Computing spectrograms from audio signals as images and using an image classification pipeline for the remainder.

2. Computing features from audio files directly as numerical vectors and applying them to a classification backend.

pyAudioProcessing includes the capability of computing spectrograms, but focusses most functionalities around the latter for building audio models. This tool contains implementations of various widely used audio feature extraction techniques, and integrates with popular scikit-learn classifiers including support vector machine (SVM), SVM radial basis function kernel (RBF), random forest, logistic regression, k-nearest neighbors (k-NN), gradient boosting, and extra trees. Audio data can be cleaned, trained, tested, and classified using pyAudioProcessing [Sin21].

Some other useful libraries for the domain of audio processing include librosa [MRL+15], spafe [Mal20], essentia [BWG+13], pyAudioAnalysis [Gia15], and paid services from service providers such as Google[1].

The use of pyAudioProcessing in the community inspires the need and growth of this software. It is referenced in a text book titled *Artificial Intelligence with Python Cookbook* published by Packt Publishing in October 2020 [Auf20]. Additionally, pyAudioProcessing is a part of specific admissions requirement for a funded PhD project at University of Portsmouth[2]. It is further referenced in this thesis paper titled "Master Thesis AI Methodologies for Processing Acoustic Signals AI Usage for Processing Acoustic Signals" [Din21], in recent research on audio processing for assessing attention levels in Attention Deficit Hyperactivity

* *Corresponding author: singhjyotika811@gmail.com*
‡ *Placemakr*

1. https://developers.google.com/learn/pathways/get-started-audio-classification

Disorder (ADHD) students [BGSR21], and more. There are thus far 16000+ downloads via pip for pyAudioProcessing with 1000+ downloads in the last month [PeP22]. As several different audio features need development, new issues are created on GitHub and contributions to the code by the open-source community are welcome to grow the tool faster.

Core Functionalities

pyAudioProcessing aims to provide an end-to-end processing solution for converting between audio file formats, visualizing time and frequency domain representations, cleaning with silence and low-activity segments removal from audio, building features from raw audio samples, and training a machine learning model that can then be used to classify unseen raw audio samples (e.g., into categories such as music, speech, etc.). This library allows the user to extract features such as Mel Frequency Cepstral Coefficients (MFCC) [CD14], Gammatone Frequency Cepstral Coefficients (GFCC) [JDHP17], spectral features, chroma features and other beat-based and cepstrum based features from audio to use with one's own classification backend or scikit-learn classifiers that have been built into pyAudioProcessing. The classifier implementation examples that are a part of this software aim to give the users a sample solution to audio classification problems and help build the foundation to tackle new and unseen problems.

pyAudioProcessing provides seven core functionalities comprising different stages of audio signal processing.

1. Converting audio files to .wav format to give the users the ability to work with different types of audio to increase compatibility with code and processes that work best with .wav audio type.

2. Audio visualization in time-series and frequency representation, including spectrograms.

3. Segmenting and removing low-activity segments from audio files for removing unwanted audio segments that are less likely to represent meaningful information.

4. Building numerical features from audio that can be used to train machine learning models. The set of features supported evolves with time as research informs new and improved algorithms.

5. Ability to export the features built with this library to use with any custom machine learning backend of the user's choosing.

6. Capability that allows users to train scikit-learn classifiers using features of their choosing directly from raw data. pyAudioProcessing

   a). runs automatic hyper-parameter tuning

   b). returns to the user the training model metrics along with cross-validation confusion matrix (a cross-validation confusion matrix is an evaluation matrix from where we can estimate the performance of the model broken down by each class/category) for model evaluation

   c). allows the user to test the created classifier with the same features used for training

7. Includes pre-trained models to provide users with baseline audio classifiers.

2. https://www.port.ac.uk/study/postgraduate-research/research-degrees/phd/explore-our-projects/detection-of-emotional-states-from-speech-and-text

Class	Metric		
	Accuracy	Precision	F1
music	97.60%	98.79%	98.19%
speech	98.80%	97.63%	98.21%

TABLE 1: *Per-class evaluation metrics for audio type (speech vs music) classification pre-trained model.*

Class	Metric		
	Accuracy	Precision	F1
music	94.60%	96.93%	95.75%
speech	97.00%	97.79%	97.39%
birds	100.00%	96.89%	98.42%

TABLE 2: *Per-class evaluation metrics for audio type (speech vs music vs bird sound) classification pre-trained model.*

Methods and Results

Pre-trained models

pyAudioProcessing offers pre-trained audio classification models for the Python community to aid in quick baseline establishment. This is an evolving feature as new datasets and classification problems gain prominence in the field.

Some of the pre-trained models include the following.

1. Audio type classifier to determine speech versus music: Trained a Support Vector Machine (SVM) classifier for classifying audio into two possible classes - music, speech. This classifier was trained using Mel Frequency Cepstral Coefficients (MFCC), spectral features, and chroma features. This model was trained on manually created and curated samples for speech and music. The per-class evaluation metrics are shown in Table 1.

2. Audio type classifier to determine speech versus music versus bird sounds: Trained Support Vector Machine (SVM) classifier for classifying audio into three possible classes - music, speech, birds. This classifier was trained using Mel Frequency Cepstral Coefficients (MFCC), spectral features, and chroma features. The per-class evaluation metrics are shown in Table 2.

3. Music genre classifier using the GTZAN [TEC01]: Trained on SVM classifier using Gammatone Frequency Cepstral Coefficients (GFCC), Mel Frequency Cepstral Coefficients (MFCC), spectral features, and chroma features to classify music into 10 genre classes - blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, rock. The per-class evaluation metrics are shown in Table 3.

These models aim to present capability of audio feature generation algorithms in extracting meaningful numeric patterns from the audio data. One can train their own classifiers using similar features and different machine learning backend for researching and exploring improvements.

Audio features

There are multiple types of features one can extract from audio. Information about getting started with audio processing is well described in [Sin19]. pyAudioProcessing allows users to compute GFCC, MFCC, other cepstral features, spectral features, temporal features, chroma features, and more. Details on how to extract these features are present in the project documentation on GitHub.

Class	Metric		
	Accuracy	Precision	F1
pop	72.36%	78.63%	75.36%
met	87.31%	85.52%	86.41%
dis	62.84%	59.45%	61.10%
blu	83.02%	72.96%	77.66%
reg	79.82%	69.72%	74.43%
cla	90.61%	86.38%	88.44%
rock	53.10%	51.50%	52.29%
hip	60.94%	77.22%	68.12%
cou	58.34%	62.53%	60.36%
jazz	78.10%	85.17%	81.48%

TABLE 3: *Per-class evaluation metrics for music genre classification pre-trained model.*

Generally, features useful in different audio prediction tasks (especially speech) include Linear Prediction Coefficients (LPC) and Linear Prediction Cepstral Coefficients (LPCC), Bark Frequency Cepstral Coefficients (BFCC), Power Normalized Cepstral Coefficients (PNCC), and spectral features like spectral flux, entropy, roll off, centroid, spread, and energy entropy.

While MFCC features find use in most commonly encountered audio processing tasks such as audio type classification, speech classification, GFCC features have been found to have application in speaker identification or speaker diarization (the process of partitioning an input audio stream into homogeneous segments according to the human speaker identity [Wik22a]). Applications, comparisons and uses can be found in [ZW13], [pat21], and [pat22].

pyAudioProcessing library includes computation of these features for audio segments of a single audio, followed by computing mean and standard deviation of all the signal segments.

Mel Frequency Cepstral Coefficients (MFCC):

The mel scale relates perceived frequency, or pitch, of a pure tone to its actual measured frequency. Humans are much better at discerning small changes in pitch at low frequencies compared to high frequencies. Incorporating this scale makes our features match more closely what humans hear. The mel-frequency scale is approximately linear for frequencies below 1 kHz and logarithmic for frequencies above 1 kHz, as shown in Figure 1. This is motivated by the fact that the human auditory system becomes less frequency-selective as frequency increases above 1 kHz.

The signal is divided into segments and a spectrum is computed. Passing a spectrum through the mel filter bank, followed by taking the log magnitude and a discrete cosine transform (DCT) produces the mel cepstrum. DCT extracts the signal's main information and peaks. For this very property, DCT is also widely used in applications such as JPEG and MPEG compressions. The peaks after DCT contain the gist of the audio information. Typically, the first 13-20 coefficients extracted from the mel cepstrum are called the MFCCs. These hold very useful information about audio and are often used to train machine learning models. The process of developing these coefficients can be seen in the form of an illustration in Figure 1. MFCC for a sample speech audio can be seen in Figure 2.

Gammatone Frequency Cepstral Coefficients (GFCC):

Another filter inspired by human hearing is the gammatone filter bank. The gammatone filter bank shape looks similar to the mel filter bank, expect the peaks are smoother than the triangular shape of the mel filters. gammatone filters are conceived to be a good approximation to the human auditory filters and are used as a front-end simulation of the cochlea. Since a human ear is the perfect receiver and distinguisher of speakers in the presence of noise or no noise, construction of gammatone filters that mimic auditory filters became desirable. Thus, it has many applications in speech processing because it aims to replicate how we hear.

GFCCs are formed by passing the spectrum through a gammatone filter bank, followed by loudness compression and DCT, as seen in Figure 3. The first (approximately) 22 features are called GFCCs. GFCCs have a number of applications in speech processing, such as speaker identification. GFCC for a sample speech audio can be seen in Figure 4.

Temporal features:

Temporal features from audio are extracted from the signal information in its time domain representations. Examples include signal energy, entropy, zero crossing rate, etc. Some sample mean temporal features can be seen in Figure 5.

Spectral features:

Spectral features on the other hand derive information contained in the frequency domain representation of an audio signal. The signal can be converted from time domain to frequency domain using the Fourier transform. Useful features from the signal spectrum include fundamental frequency, spectral entropy, spectral spread, spectral flux, spectral centroid, spectral roll-off, etc. Some sample mean spectral features can be seen in Figure 6.

Chroma features:

Chroma features are highly popular for music audio data. In Western music, the term chroma feature or chromagram closely relates to the twelve different pitch classes. Chroma-based features, which are also referred to as "pitch class profiles", are a powerful tool for analyzing music whose pitches can be meaningfully categorized (often into twelve categories : A, A#, B, C, C#, D,

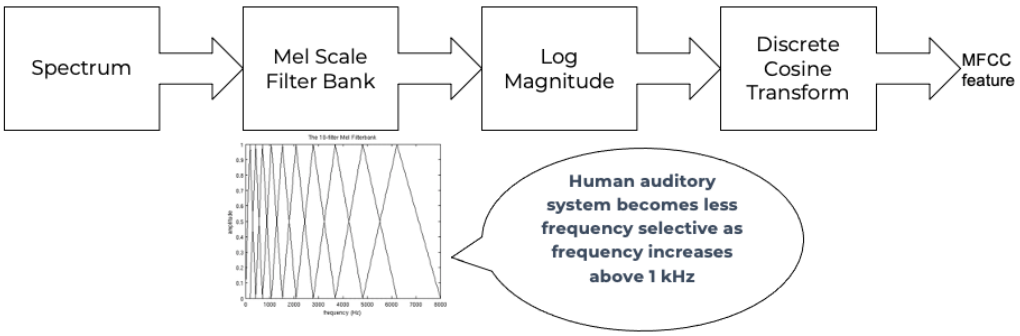


Fig. 1: *MFCC from audio spectrum.*

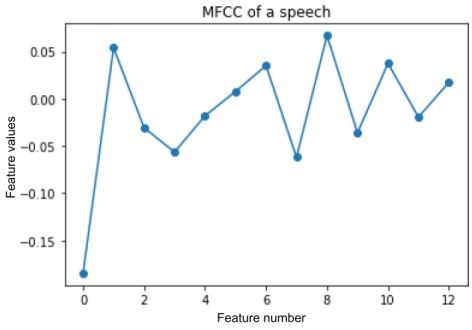


Fig. 2: *MFCC from a sample speech audio.*

D#, E, F, F#, G, G#) and whose tuning approximates to the equal-tempered scale [con22]. A prime characteristic of chroma features is that they capture the harmonic and melodic attributes of audio, while being robust to changes in timbre and instrumentation. Some sample mean chroma features can be seen in Figure 7.

Audio data cleaning/de-noising

Often times an audio sample has multiple segments present in the same signal that do not contain anything but silence or a slight degree of background noise compared to the rest of the audio. For most applications, those low activity segments make up the irrelevant information of the signal.

The audio clip shown in Figure 8 is a human saying the word "london" and represents the audio plotted in the time domain, with signal amplitude as y-axis and sample number as x-axis. The areas where the signal looks closer to zero/low in amplitude are areas where speech is absent and represents the pauses the speaker took while saying the word "london".

Figure 9 shows the spectrogram of the same audio signal. A spectrogram contains time on the x-axis and frequency of the y-axis. A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time. When applied to an audio signal, spectrograms are sometimes called sonographs, voiceprints, or voicegrams. When the data are represented in a 3D plot they may be called waterfalls. As [Wik21] mentions, spectrograms are used extensively in the fields of music, linguistics, sonar, radar, speech processing, seismology, and others. Spectrograms of audio can be used to identify spoken words phonetically, and to analyze the various calls of animals. A spectrogram can be generated by an optical spectrometer, a bank of band-pass filters, by Fourier transform or by a wavelet transform. A spectrogram is usually depicted as a heat map, i.e., as an image with the intensity shown by varying the color or brightness.

Features	boston acc	london acc
mfcc	0.765	0.412
clean+mfcc	**0.823**	**0.471**

TABLE 4: *Performance comparison on test data between MFCC feature trained model with and without cleaning.*

After applying the algorithm for signal alteration to remove irrelevant and low activity audio segments, the resultant audio's time-series plot looks like Figure 10. The spectrogram looks like Figure 11. It can be seen that the low activity areas are now missing from the audio and the resultant audio contains more activity filled regions. This algorithm removes silences as well as low-activity regions from the audio.

These visualizations were produced using pyAudioProcessing and can be produced for any audio signal using the library.

Impact of cleaning on feature formations for a classification task:

A spoken location name classification problem was considered for this evaluation. The dataset consisted of 23 samples for training per class and 17 samples for testing per class. The total number of classes is 2 - london and boston. This dataset was manually created and can be found linked in the project readme of pyAudioProcessing. For comparative purposes, the classifier is kept constant at SVM, and the parameter C is chosen based on grid search for each experiment based on best precision, recall and F1 score. Results in table 4 show the impact of applying the low-activity region removal using pyAudioProcessing prior to training the model using MFCC features.

It can be seen that the accuracies increased when audio samples were cleaned prior to training the model. This is especially useful in cases where silence or low-activity regions in the audio do not contribute to the predictions and act as noise in the signal.

Integrations

pyAudioProcessing integrates with third-party tools such as scikit-learn, matplotlib, and pydub to offer additional functionalities.

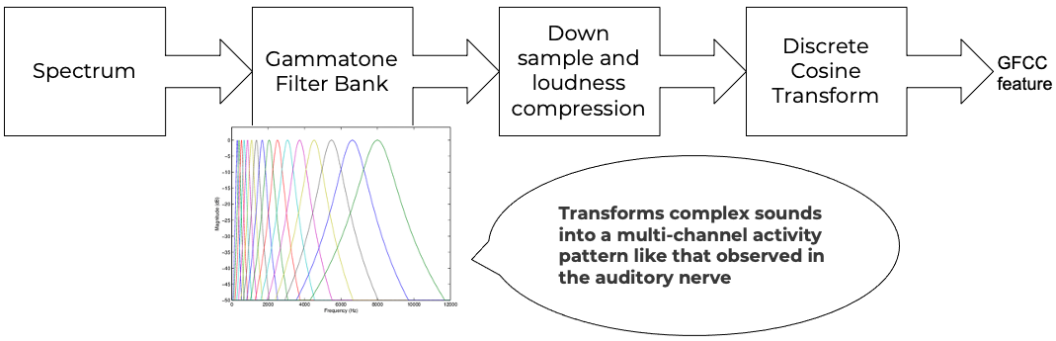


Fig. 3: *GFCC from audio spectrum.*

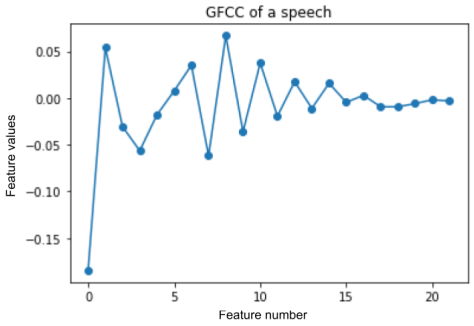


Fig. 4: *GFCC from a sample speech audio.*

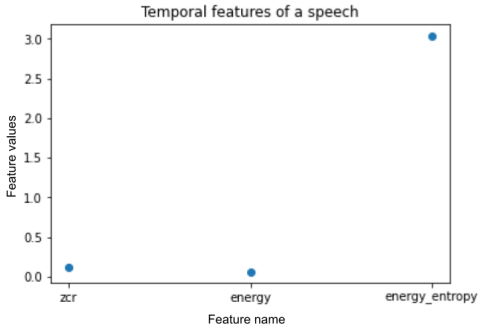


Fig. 5: *Temporal extractions from a sample speech audio.*

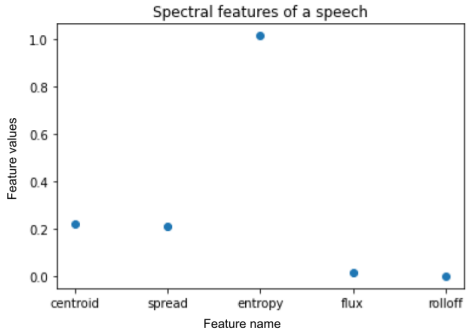


Fig. 6: *Spectral features from a sample speech audio.*

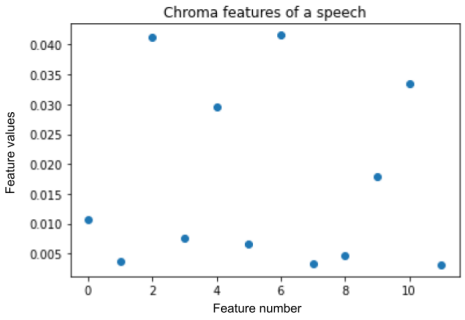


Fig. 7: *Chroma features from a sample speech audio.*

Training, classification, and evaluation:

The library contains integrations with scikit-learn classifiers for passing audio through feature extraction followed by classification directly using the raw audio samples as input. Training results include computation of cross-validation results along with hyperparameter tuning details.

Audio format conversion:

Some applications and integrations work best with .wav data format. pyAudioProcessing integrates with tools that perform format conversion and presents them as a functionality via the library.

Audio visualization:

Spectrograms are 2-D images representing sequences of spectra with time along one axis, frequency along the other, and brightness or color representing the strength of a frequency component at each time frame [Wys17]. Not only can one see whether there is more or less energy at, for example, 2 Hz vs 10 Hz, but one can also see how energy levels vary over time [PNS]. Some of the convolutional neural network architectures for images can be applied to audio signals on top of the spectrograms. This is a different route of building audio models by developing spectrograms followed by image processing. Time-series, frequency-domain, and spectrogram (both time and frequency domains) visualizations can be retrieved using pyAudioProcessing and its integrations. See figures 10 and 9 as examples.

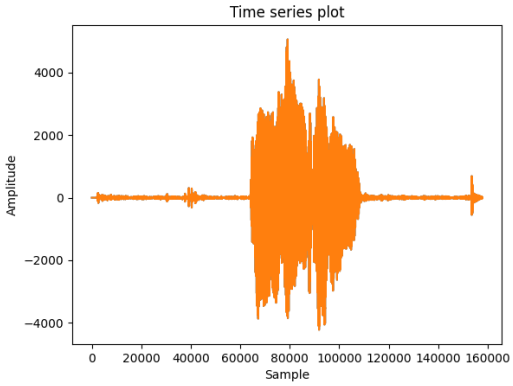


Fig. 8: *Time-series representation of speech for "london".*

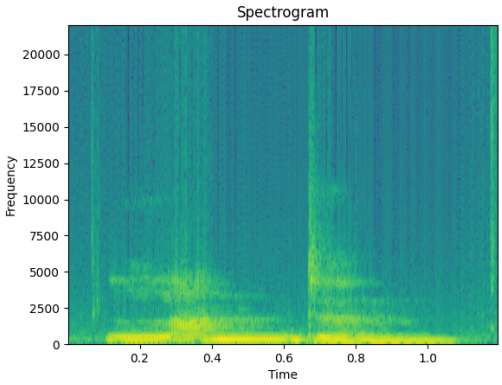


Fig. 11: *Spectrogram of cleaned speech for "london".*

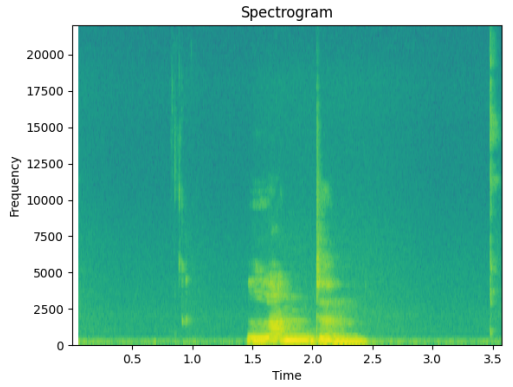


Fig. 9: *Spectrogram of speech for "london".*

Conclusion

In this paper pyAudioProcessing, an open-source Python library, is presented. The tool implements and integrates a wide range of audio processing functionalities. Using pyAudioProcessing, one can read and visualize audio signals, clean audio signals by removal of irrelevant content, build and extract complex features such as GFCC, MFCC, and other spectrum and cepstrum based features, build classification models, and use pre-built trained baseline models to classify different types of audio. Wrappers along with command-line usage examples are provided in the software's readme and wiki for giving the user a guide and the flexibility of usage. pyAudioProcessing has been used in active research around audio processing and can be used as the basis for further python-based research efforts.

pyAudioProcessing is updated frequently in order to apply enhancements and new functionalities with recent research efforts of the digital signal processing and machine learning community. Some of the ongoing implementations include additions of cepstral features such as LPCC, integration with deep learning backends, and a variety of spectrogram formations that can be used for image classification-based audio classification tasks.

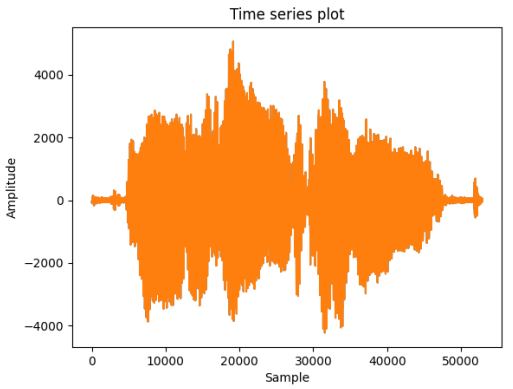


Fig. 10: *Time-series representation of cleaned speech for "london".*

References

[Auf20] Ben Auffarth. *Artificial Intelligence with Python Cookbook*. Packt Publishing, 10 2020.

[BGSR21] Srivi Balaji, Meghana Gopannagari, Svanik Sharma, and Preethi Rajgopal. Developing a machine learning algorithm to assess attention levels in adh students in a virtual learning setting using audio and video processing. *International Journal of Recent Technology and Engineering (IJRTE)*, 10, 5 2021. doi:10.35940/ijrte.A5965.0510121.

[BWG+13] Dmitry Bogdanov, N Wack, Emilia Gómez, Sankalp Gulati, Perfecto Herrera, Oscar Mayor, G Roma, Justin Salamon, Jose Zapata, and Xavier Serra. Essentia: an audio analysis library for music information retrieval. 11 2013.

[CD14] Paresh M. Chauhan and Nikita P. Desai. Mel frequency cepstral coefficients (mfcc) based speaker identification in noisy environment using wiener filter. In *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*, pages 1–5, 2014. doi:10.1109/ICGCCEE.2014.6921394.

[con22] Wikipedia contributors. Chroma feature — wikipedia the free encyclopedia, 2022. Online; accessed 18-May-2022. URL: https://en.wikipedia.org/w/index.php?title=Chroma_feature&oldid=1066722932.

[Din21] Vincent Dinger. *Master Thesis KI Methodiken fÃ¼r die Verarbeitung akustischer Signale AI Usage for Processing Acoustic Signals*. PhD thesis, Kaiserslautern University of Applied Sciences, 03 2021. doi:10.13140/RG.2.2.15872.97287.

[Gia15] Theodoros Giannakopoulos. pyaudioanalysis: An open-source python library for audio signal analysis. *PloS one*, 10(12), 2015. doi:10.1371/journal.pone.0144610.

[JDHP17] Medikonda Jeevan, Atul Dhingra, M. Hanmandlu, and Bijaya Panigrahi. *Robust Speaker Verification Using GFCC Based i-Vectors*, volume 395, pages 85–91. Springer, 10 2017. doi:10.1007/978-81-322-3592-7_9.

[JS21] Jyotika Singh. Social Media Analysis using Natural Language Processing Techniques. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 20th Python in Science Conference*, pages 52 – 58, 2021. URL: http://conference.scipy.org/proceedings/scipy2021/

pdfs/jyotika_singh.pdf, doi:10.25080/majora-1b6fd038-009.

[Mal20] Ayoub Malek. spafe/spafe: 0.1.2, April 2020. URL: https://github.com/SuperKogito/spafe.

[MRL+15] Brian McFee, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, volume 8, 2015. doi:10.5281/zenodo.4792298.

[pat21] Method for optimizing media and marketing content using cross-platform video intelligence, 2021. URL: https://patents.google.com/patent/US10949880B2/en.

[pat22] Media and marketing optimization with cross platform consumer and content intelligence, 2022. URL: https://patents.google.com/patent/US20210201349A1/en.

[PeP22] PePy. PePy download statistics, 2022. URL: https://pepy.tech/project/pyAudioProcessing.

[PNS] PNSN. What is a spectrogram? URL: https://pnsn.org/spectrograms/what-is-a-spectrogram#.

[Sin19] Jyotika Singh. An introduction to audio processing and machine learning using python, 2019. URL: https://opensource.com/article/19/9/audio-processing-machine-learning-python.

[Sin21] Jyotika Singh. jsingh811/pyAudioProcessing: Audio processing, feature extraction and classification, July 2021. URL: https://github.com/jsingh811/pyAudioProcessing, doi:10.5281/zenodo.5121041.

[TEC01] George Tzanetakis, Georg Essl, and Perry Cook. Automatic musical genre classification of audio signals, 2001. URL: http://ismir2001.ismir.net/pdf/tzanetakis.pdf.

[Wik21] Wikipedia contributors. Spectrogram — Wikipedia, the free encyclopedia, 2021. [Online; accessed 19-July-2021]. URL: https://en.wikipedia.org/w/index.php?title=Spectrogram&oldid=1031156666.

[Wik22a] Wikipedia contributors. Speaker diarisation — Wikipedia, the free encyclopedia, 2022. [Online; accessed 23-June-2022]. URL: https://en.wikipedia.org/w/index.php?title=Speaker_diarisation&oldid=1090834931.

[Wik22b] Wikipedia contributors. Word embedding — Wikipedia, the free encyclopedia, 2022. [Online; accessed 23-June-2022]. URL: https://en.wikipedia.org/w/index.php?title=Word_embedding&oldid=1091348337.

[Wys17] Lonce Wyse. Audio spectrogram representations for processing with convolutional neural networks. 06 2017.

[ZW13] Xiaojia Zhao and DeLiang Wang. Analyzing noise robustness of mfcc and gfcc features in speaker identification. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7204–7208, 2013. doi:10.1109/ICASSP.2013.6639061.

Phylogeography: Analysis of genetic and climatic data of SARS-CoV-2

Aleksandr Koshkarov[‡§¶*], Wanlin Li[‡¶], My-Linh Luu[‖], Nadia Tahiri[‡]

Abstract—Due to the fact that the SARS-CoV-2 pandemic reaches its peak, researchers around the globe are combining efforts to investigate the genetics of different variants to better deal with its distribution. This paper discusses phylogeographic approaches to examine how patterns of divergence within SARS-CoV-2 coincide with geographic features, such as climatic features. First, we propose a python-based bioinformatic pipeline called **aPhylogeo** for phylogeographic analysis written in Python 3 that help researchers better understand the distribution of the virus in specific regions via a configuration file, and then run all the analysis operations in a single run. In particular, the aPhylogeo tool determines which parts of the genetic sequence undergo a high mutation rate depending on geographic conditions, using a sliding window that moves along the genetic sequence alignment in user-defined steps and a window size. As a Python-based cross-platform program, aPhylogeo works on Windows®, MacOS X® and GNU/Linux. The implementation of this pipeline is publicly available on GitHub (https://github.com/tahiri-lab/aPhylogeo). Second, we present an example of analysis of our new aPhylogeo tool on real data (SARS-CoV-2) to understand the occurrence of different variants.

Index Terms—Phylogeography, SARS-CoV-2, Bioinformatics, Genetic, Climatic Condition

Introduction

The global pandemic caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) is at its peak and more and more variants of SARS-CoV-2 were described over time. Among these, some are considered variants of concern (VOC) by the World Health Organization (WHO) due to their impact on global public health, such as Alpha (B.1.1.7), Beta (B.1.351), Gamma (P.1), Delta (B.1.617.2), and Omicron (B.1.1.529) [CRA+22]. Although significant progress was made in vaccine development and mass vaccination is being implemented in many countries, the continued emergence of new variants of SARS-CoV-2 threatens to reverse the progress made to date. Researchers around the world collaborate to better understand the genetics of the different variants, along with the factors that influence the epidemiology of this infectious disease. Genetic studies of the different variants contributed to the development of vaccines to better combat the spread of the virus. Studying the factors (e.g., environment, host, agent of transmission) that influence epidemiology helps us to limit the continued spread of infection and prepare for the future re-emergence of diseases caused by subtypes of coronavirus [LFZK06]. However, few studies report associations between environmental factors and the genetics of different variants. Different variants of SARS-CoV-2 are expected to spread differently depending on geographical conditions, such as the meteorological parameters. The main objective of this study is to find clear correlations between genetics and geographic distribution of different variants of SARS-CoV-2.

Several studies showed that COVID-19 cases and related climatic factors correlate significantly with each other ([OCFC20], [SDdPS+20], and [SMVS+22]). Oliveiros et al. [OCFC20] reported a decrease in the rate of SARS-CoV-2 progression with the onset of spring and summer in the northern hemisphere. Sobral et al. [SDdPS+20] suggested a negative correlation between mean temperature by country and the number of SARS-CoV-2 infections, along with a positive correlation between rainfall and SARS-CoV-2 transmission. This contrasts with the results of the study by Sabarathinam et al. [SMVS+22], which showed that an increase in temperature led to an increase in the spread of SARS-CoV-2. The results of Chen et al. [CPK+21] imply that a country located 1000 km closer to the equator can expect 33% fewer cases of SARS-CoV-2 per million population. Some virus variants may be more stable in environments with specific climatic factors. Sabarathinam et al. [SMVS+22] compared mutation patterns of SARS-CoV-2 with time series of changes in precipitation, humidity, and temperature. They suggested that temperatures between 43°F and 54°F, humidity of 67-75%, and precipitation of 2-4 mm may be the optimal environment for the transition of the mutant form from D614 to G614.

In this study, we examine the geospatial lineage of SARS-CoV-2 by combining genetic data and metadata from associated sampling locations. Thus, an association between genetics and the geographic distribution of SARS-CoV-2 variants can be found. We focus on developing a new algorithm to find relationships between a reference tree (i.e., a tree of geographic species distributions, a temperature tree, a habitat precipitation tree, or others) with their genetic compositions. This new algorithm can help find which genes or which subparts of a gene are sensitive or favorable to a given environment.

* *Corresponding author: Nadia.Tahiri@USherbrooke.ca*
‡ *Department of Computer Science, University of Sherbrooke, Sherbrooke, QC J1K2R1, Canada*
§ *Center of Artificial Intelligence, Astrakhan State University, Astrakhan, 414056, Russia*
¶ *Contributed equally*
‖ *Department of Computer Science, University of Quebec at Montreal, Montreal, QC, Canada*

Problem statement and proposal

Phylogeography is the study of the principles and processes that govern the distribution of genealogical lineages, particularly at the intraspecific level. The geographic distribution of species is often correlated with the patterns associated with the species' genes ([A+00] and [KM02]). In a phylogeographic study, three major processes should be considered (see [Nag92] for more details), which are:

1) Genetic drift is the result of allele sampling errors. These errors are due to generational transmission of alleles and geographical barriers. Genetic drift is a function of the size of the population. Indeed, the larger the population, the lower the genetic drift. This is explained by the ability to maintain genetic diversity in the original population. By convention, we say that an allele is fixed if it reaches the frequency of 100%, and that it is lost if it reaches the frequency of 0%.
2) Gene flow or migration is an important process for conducting a phylogeographic study. It is the transfer of alleles from one population to another, increasing intrapopulation diversity and decreasing interpopulation diversity.
3) There are many selections in all species. Here we indicate the two most important of them, if they are essential for a phylogeographic study. (a) Sexual selection is a phenomenon resulting from an attractive characteristic between two species. Therefore, this selection is a function of the size of the population. (b) Natural selection is a function of fertility, mortality, and adaptation of a species to a habitat.

Populations living in different environments with varying climatic conditions are subject to pressures that can lead to evolutionary divergence and reproductive isolation ([OS98] and [Sch01]). Phylogeny and geography are then correlated. This study, therefore, aims to present an algorithm to show the possible correlation between certain genes or gene fragments and the geographical distribution of species.

Most studies in phylogeography consider only genetic data without directly considering climatic data. They indirectly take this information as a basis for locating the habitat of the species. We have developed the first version of a phylogeography that integrates climate data. The sliding window strategy provides more robust results, as it particularly highlights the areas sensitive to climate adaptation.

Methods and Python scripts

In order to achieve our goal, we designed a workflow and then developed a script in Python version 3.9 called **aPhylogeo** for phylogeographic analysis (see [LLKT22] for more details). It interacts with multiple bioinformatic programs, taking climatic data and nucleotide data as input, and performs multiple phylogenetic analyses on nucleotide sequencing data using a sliding window approach. The process is divided into three main steps (see Figure 1).

The first step involves collecting data to search for quality viral sequences that are essential for the conditions of our results. All sequences were retrieved from the NCBI Virus website (National Center for Biotechnology Information, https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/). In total, 20 regions were selected to represent 38 gene sequences of SARS-CoV-2. After collecting genetic data, we extracted 5 climatic factors for the 20 regions, i.e., Temperature, Humidity, Precipitation, Wind speed, and Sky surface shortwave downward irradiance. This data was obtained from the NASA website (https://power.larc.nasa.gov/).

In the second step, trees are created with climatic data and genetic data, respectively. For climatic data, we calculated the dissimilarity between each pair of variants (i.e., from different climatic conditions), resulting in a symmetric square matrix. From this matrix, the neighbor joining algorithm was used to construct the climate tree. The same approach was implemented for genetic data. Using nucleotide sequences from the 38 SARS-CoV-2 lineages, phylogenetic reconstruction is repeated to construct genetic trees, considering only the data within a window that moves along the alignment in user-defined steps and window size (their length is denoted by the number of base pairs (bp)).

In the third step, the phylogenetic trees constructed in each sliding window are compared to the climatic trees using the Robinson and Foulds (RF) topological distance [RF81]. The distance was normalized by $2n-6$, where n is the number of leaves (i.e., taxa). The proposed approach considers bootstrapping. The implementation of sliding window technology provides a more accurate identification of regions with high gene mutation rates.

As a result, we highlighted a correlation between parts of genes with a high rate of mutations depending on the geographic distribution of viruses, which emphasizes the emergence of new variants (i.e., Alpha, Beta, Delta, Gamma, and Omicron).

The creation of phylogenetic trees, as mentioned above, is an important part of the solution and includes the main steps of the developed pipeline. This function is intended for genetic data. The main parameters of this part are as follows:

```
def create_phylo_tree(gene,
                      window_size,
                      step_size,
                      bootstrap_threshold,
                      rf_threshold,
                      data_names):

   number_seq = align_sequence(gene)
   sliding_window(window_size, step_size)
   ...
   for file in files:
      try:
            ...
            create_bootstrap()
            run_dnadist()
            run_neighbor()
            run_consense()
            filter_results(gene,
                           bootstrap_threshold,
                           rf_threshold,
                           data_names,
                           number_seq,
                           file))
            ...
      except Exception as error:
            raise
```

This function takes gene data, window size, step size, bootstrap threshold, threshold for the Robinson and Foulds distance, and data names as input parameters. Then the function sequentially connects the main steps of the pipeline: *align_sequence(gene)*, *sliding_window(window_size, step_size)*, *create_bootstrap()*, *run_dnadist()*, *run_neighbor()*, *run_consense()*, and *filter_results* with parameters. As a result, we obtain a phylogenetic tree (or several trees), which is written to a file.

We have created a function (*create_tree*) to create the climate trees. The function is described as follow:

```
def create_tree(file_name, names):
    for i in range(1, len(names)):

        create_matrix(file_name,
                      names[0],
                      names[i],
                      "infile")

        os.system("./exec/neighbor " +
                  "< input/input.txt")

        subprocess.call(["mv",
                         "outtree",
                         "intree"])

        subprocess.call(["rm",
                         "infile",
                         "outfile"])

        os.system("./exec/consense "+
                  "< input/input.txt")

        newick_file = names[i].replace(" ", "_") +
                      "_newick"

        subprocess.call(["rm",
                         "outfile"])

        subprocess.call(["mv",
                         "outtree",
                         newick_file])
```

The sliding window strategy can detect genetic fragments depending on environmental parameters, but this work requires time-consuming data preprocessing and the use of several bioinformatics programs. For example, we need to verify that each sequence identifier in the sequencing data always matches the corresponding metadata. If samples are added or removed, we need to check whether the sequencing dataset matches the metadata and make changes accordingly. In the next stage, we need to align the sequences (multiple sequence alignment, MSA) and integrate all step by step into specific software such as MUSCLE [Edg04], Phylip package (i.e. Seqboot, DNADist, Neighbor, and Consense) [Fel05], RF [RF81], and raxmlHPC [Sta14]. The use of each software requires expertise in bioinformatics. In addition, the intermediate analysis steps inevitably generate many files, the management of which not only consumes the time of the biologist, but is also subject to errors, which reduces the reproducibility of the study. At present, there are only a few systems designed to automate the analysis of phylogeography. In this context, the development of a computer program for a better understanding of the nature and evolution of coronavirus is essential for the advancement of clinical research.

The following sliding window function illustrates moving the sliding window through an alignment with window size and step size as parameters. The first 11 characters are allocated to species names, plus a space.

```
def sliding_window(window_size=0, step=0):
   try:
      f = open("infile", "r")
      ...
      # slide the window along the sequence
      start  = 0
      fin = start + window_size
      while fin <= longueur:
         index = 0
         with open("out", "r") as f, ... as out:
               ...
               for line in f:
                  if line != "\n":
                     espece = list_names[index]
                     nb_espace = 11 - len(espece)
                     out.write(espece)
                     for i in range(nb_espace):
                           out.write(" ")
                     out.write(line[debut:fin])
                     index = index + 1
         out.close()
         f.close()
         start = start + step
         fin = fin + step
   except:
      print("An error occurred.")
```

Algorithmic complexity

The complexity of the algorithm described in the previous section depends on the complexity of the various external programs used and the number of windows that the alignment can contain, plus one for the total alignment that the program will process.

Recall the different complexities of the different external programs used in the algorithm:

- SeqBoot program: $\mathcal{O}(r \times n \times SA)$
- DNADist program: $\mathcal{O}(n^2)$
- Neighbor program: $\mathcal{O}(n^3)$
- Consense program: $\mathcal{O}(r \times n^2)$
- RaxML program: $\mathcal{O}(e \times n \times SA)$
- RF program: $\mathcal{O}(n^2)$,

where n is a number of species (or taxa), r is a number of replicates, SA is a size of the multiple sequence alignment (MSA), and e is a number of refinement steps performed by the RaxML algorithm. For all $SA \in N^*$ and for all $WS, S \in N$, the number of windows can be evaluated as follow (Eq. 1):

$$nb = \left\lfloor \frac{SA - WS}{S} + 1 \right\rfloor, \tag{1}$$

where WS is a window size, and S is a step.

Dataset

The following two principles were applied to select the samples for analysis.

1) **Selection of SARS-CoV-2 Pango lineages that are dispersed in different phylogenetic clusters whenever possible.**

The Pango lineage nomenclature system is hierarchical and fine-scaled and is designed to capture the leading edge of pandemic transmission. Each Pango lineage aims to define an epidemiologically relevant phylogenetic cluster, for instance, an introduction into a distinct geographic area with evidence of onward transmission [RHO+20]. From one side, Pango lineages signify groups or clusters of infections with shared ancestry. If the entire pandemic can be thought of as a vast branching tree of transmission, then the Pango lineages represent individual branches within that tree. From another side, Pango lineages are intended to highlight epidemiologically relevant events, such as the appearance of the virus in a new location, a rapid increase in the number of cases, or the evolution of viruses with new phenotypes [OSU+21]. Therefore, to have some sequence diversity in the selected samples, we avoided selecting lineages belonging to the same or similar phylogenetic clusters. For example, among

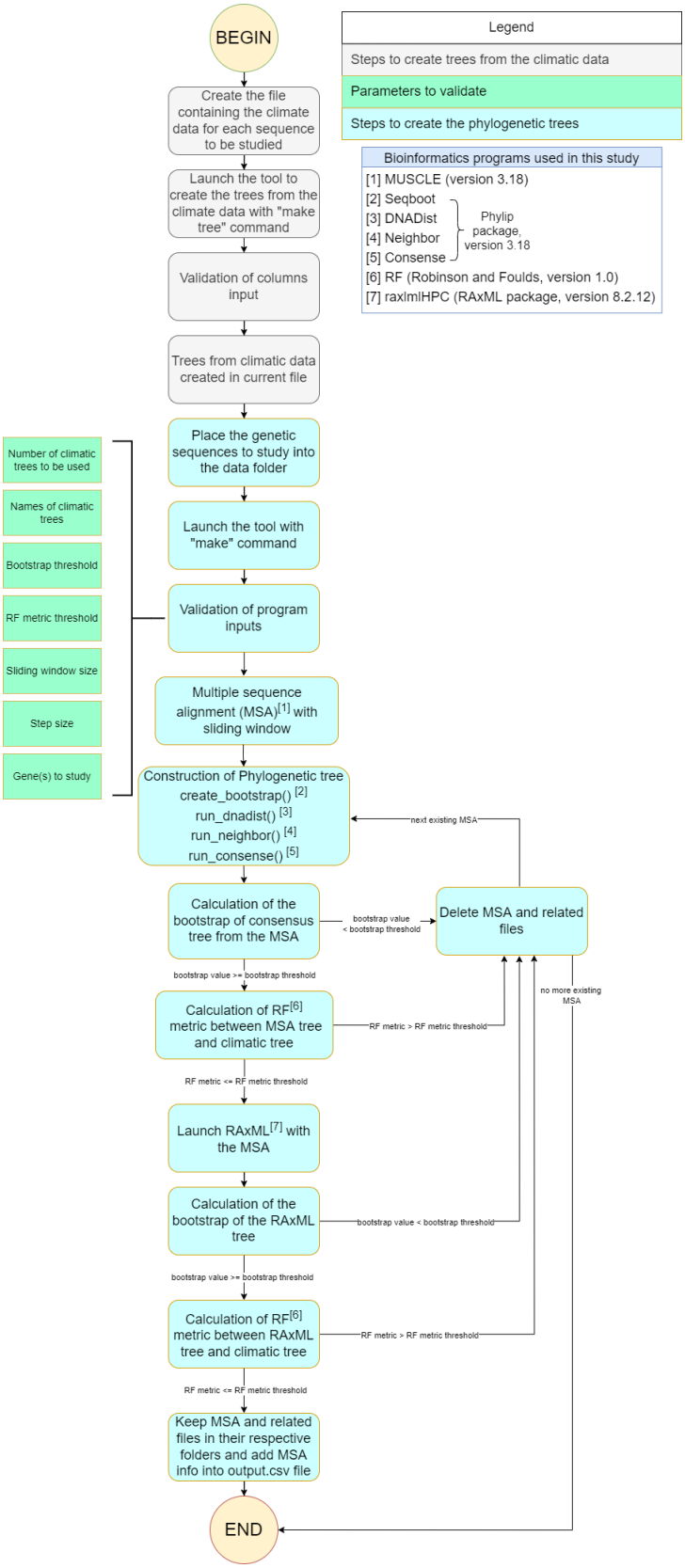


Fig. 1: *The workflow of the algorithm. The operations within this workflow include several blocks. The blocks are highlighted by three different colors. The first block (grey color) is responsible for creating the trees based on the climate data. The second block (green color) performs the function of input parameter validation. The third block (blue color) allows the creation of phylogenetic trees. This is the most important block and the basis of this study, through the results of which the user receives the output data with the necessary calculations.*

C.36, C.36.1, C.36.2, C.36.3 and C.36.3.1, only C.36 was used as a sample for analysis.

2) **Selection of the lineages that are clearly dominant in a particular region compared to other regions.**

Through significant advances in the generation and exchange of SARS-CoV-2 genomic data in real time, international spread of lineages is tracked and recorded on the website (cov-lineages.org/global_report.html) [OHP+21]. Based on the statistical information provided by the website, our study focuses on SARS-CoV-2 lineages that were first identified (Earliest date) and widely disseminated in a particular country (Most common country) during a certain period (Table 1).

We list four examples of the distribution of a set of lineages:

- Both lineages A.2.3 and B.1.1.107 have 100% distribution in the United Kingdom. Both lineages D.2 and D.3 have 100% distribution in Australia. B.1.1.172, L.4 and P.1.13 have 100% distribution in the United States. Finally, AH.1, AK.2, C.7 have 100% distribution in Switzerland, Germany, and Denmark, respectively.
- The country with the widest distribution of L.2 is the Netherlands (77.0%), followed by Germany (19.0%). Due to a 58% difference in the distribution of L.2 between the two locations, we consider the Netherlands as the main distribution country of L.2 and, therefore, it was selected as a sample.
- Similarly, the most predominant country of distribution of C.37 is Peru (44%), followed by Chile (19.0%), with a difference of 25%. Among all samples of this study, C.37 was the lineage with the least difference in distribution percentage between the two countries. Considering the need to increase the diversity of the geographical distribution of the samples, C.37 was also selected.
- In contrast, the distribution of C.6 is 17.0% in France, 14.0% in Angola, 13.0% in Portugal, and 8.0% in Switzerland, and we concluded that C.6 does not show a tendency in terms of geographic distribution and, therefore, was not included as a sample for analysis.

In accordance with the above principles, we selected 38 lineages with regional characteristics for further study. Based on location information, complete nucleotide sequencing data for these 38 lineages was collected from the NCBI Virus website (https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/). In the case of the availability of multiple sequencing results for the same lineage in the same country, we selected the sequence whose collection date was closest to the earliest date presented. If there are several sequencing results for the same country on the same date, the sequence with the least number of ambiguous characters (N per nucleotide) is selected (Table 1).

Based on the sampling locations (consistent with the most common country, but accurate to specific cities) of each lineage sequence in Table 1, combined with the time when the lineage was first discovered, we obtained data on climatic conditions at the time each lineage was first discovered. The meteorological parameters include Temperature at 2 meters, Specific humidity at 2 meters, Precipitation corrected, Wind speed at 10 meters, and All sky surface shortwave Downward irradiance. The daily data for the above parameters were collected from the NASA website (https://power.larc.nasa.gov/). Considering that the spread of the virus in a country and the data statistics are time-consuming, we collected climatological data for the three days before the earliest reporting date corresponding to each lineage and averaged them for analysis (Fig. 2).

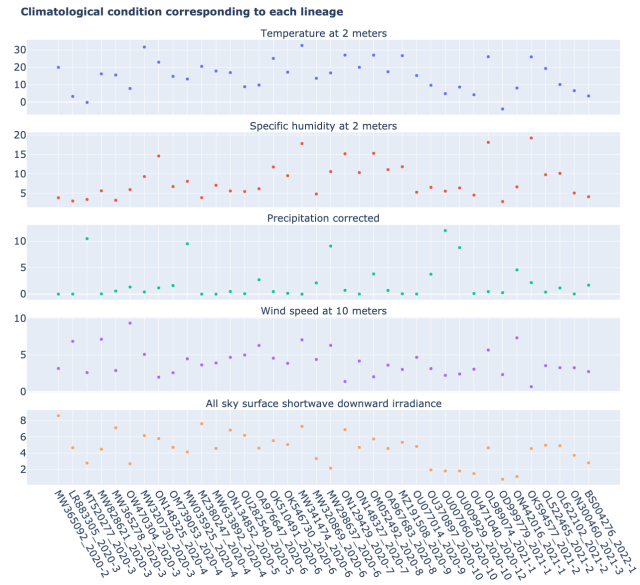


Fig. 2: Climatic conditions of each lineage in most common country at the time of first detection. The climate factors involved include Temperature at 2 meters (C), Specific humidity at 2 meters (g/kg), Precipitation corrected (mm/day), Wind speed at 10 meters (m/s), and All sky surface shortwave downward irradiance $(kW-hr/m^2/day)$.

Although the selection of samples was based on the phylogenetic cluster of lineage and transmission, most of the sites involved represent different meteorological conditions. As shown in Figure 2, the 38 samples involved temperatures ranging from -4 C to 32.6 C, with an average temperature of 15.3 C. The Specific humidity ranged from 2.9 g/kg to 19.2 g/kg with an average of 8.3 g/kg. The variability of Wind speed and All sky surface shortwave downward irradiance was relatively small across samples compared to other parameters. The Wind speed ranged from 0.7 m/s to 9.3 m/s with an average of 4.0 m/s, and All sky surface shortwave downward irradiance ranged from 0.8 kW-hr/m2/day to 8.6 kW-hr/m2/day with an average of 4.5 kW-hr/m2/day. In contrast to the other parameters, 75% of the cities involved receive less than 2.2 mm of precipitation per day, and only 5 cities have more than 5 mm of precipitation per day. The minimum precipitation is 0 mm/day, the maximum precipitation is 12 mm/day, and the average value is 2.1 mm/day.

Results

In this section, we describe the results obtained on our dataset (see Data section) using our new algorithm (see Method section).

The size of the sliding window and the advanced step for the sliding window play an important role in the analysis. We restricted our conditions to certain values. For comparison, we applied five combinations of parameters (window size and step size) to the same dataset. These include the choice of different window sizes (20bp, 50bp, 200bp) and step sizes (10bp, 50bp, 200bp). These combinations of window sizes and steps provide an opportunity to have three different movement strategies (overlapping, non-overlapping, with gaps). Here we fixed the pair (window

Lineage	Most Common Country	Earliest Date	Sequence Accession
A.2.3	United Kingdom 100.0%	2020-03-12	OW470304.1
AE.2	Bahrain 100.0%	2020-06-23	MW341474
AH.1	Switzerland 100.0%	2021-01-05	OD999779
AK.2	Germany 100.0%	2020-09-19	OU077014
B.1.1.107	United Kingdom 100.0%	2020-06-06	OA976647
B.1.1.172	USA 100.0%	2020-04-06	MW035925
BA.2.24	Japan 99.0%	2022-01-27	BS004276
C.1	South Africa 93.0%	2020-04-16	OM739053.1
C.7	Denmark 100.0%	2020-05-11	OU282540
C.17	Egypt 69.0%	2020-04-04	MZ380247
C.20	Switzerland 85.0%	2020-10-26	OU007060
C.23	USA 90.0%	2020-05-11	ON134852
C.31	USA 87.0%	2020-08-11	OM052492
C.36	Egypt 34.0%	2020-03-13	MW828621
C.37	Peru 43.0%	2021-02-02	OL622102
D.2	Australia 100.0%	2020-03-19	MW320730
D.3	Australia 100.0%	2020-06-14	MW320869
D.4	United Kingdom 80.0%	2020-08-13	OA967683
D.5	Sweden 65.0%	2020-10-12	OU370897
Q.2	Italy 99.0%	2020-12-15	OU471040
Q.3	USA 99.0%	2020-07-08	ON129429
Q.6	France 92.0%	2021-03-02	ON300460
Q.7	France 86.0%	2021-01-29	ON442016
L.2	Netherlands 73.0%	2020-03-23	LR883305
L.4	USA 100.0%	2020-06-29	OK546730
N.1	USA 91.0%	2020-03-25	MT520277
N.3	Argentina 96.0%	2020-04-17	MW633892
N.4	Chile 92.0%	2020-03-25	MW365278
N.6	Chile 98.0%	2020-02-16	MW365092
N.7	Uruguay 100.0%	2020-06-18	MW298637
N.8	Kenya 94.0%	2020-06-23	OK510491
N.9	Brazil 96.0%	2020-09-25	MZ191508
M.2	Switzerland 90.0%	2020-10-26	OU009929
P.1.7.1	Peru 94.0%	2021-02-07	OK594577
P.1.13	USA 100.0%	2021-02-24	OL522465
P.2	Brazil 58.0%	2020-04-13	ON148325
P.3	Philippines 83.0%	2021-01-08	OL989074
P.7	Brazil 71.0%	2020-07-01	ON148327

TABLE 1: *SARS-CoV-2 lineages analyzed. The lineage assignments covered in the table were last updated on March 1, 2022. Among all Pango lineages of SARS-CoV-2, 38 lineages were analyzed. Corresponding sequencing data were found in the NCBI database based on the date of earliest detection and country of most common. The table also marks the percentage of the virus in the most common country compared to all countries where the virus is present.*

size, step size) at some values (20, 10), (20, 50), (50, 50), (200, 50) and (200, 200).

1) **Robinson and Foulds baseline and bootstrap threshold**: the phylogenetic trees constructed in each sliding window are compared to the climatic trees using the Robinson and Foulds topological distance (the RF distance). We defined the value of the RF distance obtained for regions without any mutations as the baseline. Although different sample sizes and sample sequence characteristics can cause differences in the baseline, however, regions without any mutation are often accompanied by very low bootstrap values. Using the distribution of bootstrap values and combining it with validation of alignment visualization, we confirmed that the RF baseline value in this study was 50, and the bootstrap values corresponding to this baseline were smaller than 10.
2) **Sliding window**: the implementation of sliding window technology with bootstrap threshold provides a more accurate identification of regions with high gene mutation rates. Figure 3 shows the general pattern of the RF distance changes over alignment windows with different climate conditions on bootstrap values greater than 10. The trend of RF values variation under different climatic conditions does not vary much throughout this whole sequence sliding window scan, which may be related to the correlation between climatic factors (Wind Speed, Downward Irradiance, Precipitation, Humidity, Temperature). Windows starting from or containing position (28550bp) were screened in all five scans for different combinations of window size and step size. The window formed from position 29200bp to position 29470bp is screened out in all four scans except for the combination of 50bp window size with 50bp step size. As Figure 3 shows, if there are gaps in the scan (window size: 20bp, step size: 50bp), some potential mutation windows are not screened compared to other movement strategies because the sequences of the gap part are not computed by the algorithm. In addition, when the window size is small, the capture of the window mutation signal becomes more sensitive, especially when the number of samples is small. At this time, a single base change in a single sequence can cause a change in the value of the RF distance. Therefore, high quality sequencing data is required to prevent errors

caused by ambiguous characters (N in nucleotide) on the RF distance values. In cases where a larger window size (200bp) is selected, the overlapping movement strategy (window size: 200bp, step size: 50bp) allows the signal of base mutations to be repeatedly verified and enhanced in adjacent window scans compared to the non-overlapping strategy (window size: 200bp, step size: 200bp). In this situation, the range of the RF distance values is relatively large, and the number of windows eventually screened is relatively greater. Due to the small number of the SARS-CoV-2 lineages sequences that we analyzed in this study, we chose to scan the alignment sequences with a larger window and overlapping movement strategy for further analysis (window size: 200bp, step size: 50bp).

3) **Comparaison between genetic trees and climatic trees**: the RF distance quantified the difference between a phylogenetic tree constructed in specific sliding windows and a climatic tree constructed in corresponding climatic data. Relatively low RF distance values represent relatively more similarity between the phylogenetic tree and the climatic tree. With our algorithm based on the sliding window technique, regions with high mutation rates can be identified (Fig 4). Subsequently, we compare the RF values of these regions. In cases where there is a correlation between the occurrence of mutations and the climate factors studied, the regions with relatively low RF distance values (the alignment position of 15550bp – 15600bp and 24650bp-24750bp) are more likely to be correlated with climate factors than the other loci screened for mutations.

In addition, we can state that we have made an effort to make our tool as independent as possible of the input data and parameters. Our pipeline can also be applied to phylogeographic studies of other species. In cases where it is determined (or assumed) that the occurrence of a mutation is associated with certain geographic factors, our pipeline can help to highlight mutant regions and specific mutant regions within them that are more likely to be associated with that geographic parameter. Our algorithm can provide a reference for further biological studies.

Conclusions and future work

In this paper, a bioinformatics pipeline for phylogeographic analysis is designed to help researchers better understand the distribution of viruses in specific regions using genetic and climate data. We propose a new algorithm called **aPhylogeo** [LLKT22] that allows the user to quickly and intuitively create trees from genetic and climate data. Using a sliding window, the algorithm finds specific regions on the viral genetic sequences that can be correlated to the climatic conditions of the region. To our knowledge, this is the first study of its kind that incorporates climate data into this type of study. It aims to help the scientific community by facilitating research in the field of phylogeography. Our solution runs on Windows®, MacOS X® and GNU/Linux and the code is freely available to researchers and collaborators on GitHub (https://github.com/tahiri-lab/aPhylogeo).

As a future work on the project, we plan to incorporate the following additional features:

1) We can handle large amounts of data, especially when considering many countries and longer time periods (dates). In addition, since the size of the sliding window and the forward step play an important role in the analysis, we need to perform several tests to choose the best combination of parameters. In this case, it is important to provide the faster performance of this solution, and we plan to adapt the code to parallelize the computations. In addition, we intend to use the resources of Compute Canada and Compute Quebec for these high load calculations.
2) To enable further analysis of this topic, it would be interesting to relate the results obtained, especially the values obtained from the best positions of the multiple sequence alignments, to the dimensional structure of the proteins, or to the map of the selective pressure exerted on the indicated alignment fragments.
3) We can envisage a study that would consist in selecting only different phenotypes of a single species, for example, *Homo Sapiens*, in different geographical locations. In this case, we would have to consider a larger geographical area in order to significantly increase the variation of the selected climatic parameters. This type of research would consist in observing the evolution of the genes of the selected species according to different climatic parameters.
4) We intend to develop a website that can help biologists, ecologists and other interested professionals to perform calculations in their phylogeography projects faster and easier. We plan to create a user-friendly interface with the input of the necessary initial parameters and the possibility to save the results (for example, by sending them to an email). These results will include calculated parameters and visualizations.

Acknowledgements

The authors thank SciPy conference and reviewers for their valuable comments on this paper. This work was supported by Natural Sciences and Engineering Research Council of Canada and the University of Sherbrooke grant.

References

[A+00] John C Avise et al. *Phylogeography: the history and formation of species*. Harvard University Press, 2000. doi:10.1093/icb/41.1.134.

[CPK+21] Simiao Chen, Klaus Prettner, Michael Kuhn, Pascal Geldsetzer, Chen Wang, Till Bärnighausen, and David E Bloom. Climate and the spread of covid-19. *Scientific Reports*, 11(1):1–6, 2021. doi:10.1038/s41598-021-87692-z.

[CRA+22] Marco Cascella, Michael Rajnik, Abdul Aleem, Scott C Dulebohn, and Raffaela Di Napoli. Features, evaluation, and treatment of coronavirus (covid-19). *Statpearls [internet]*, 2022.

[Edg04] Robert C Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC bioinformatics*, 5(1):1–19, 2004. doi:10.1186/1471-2105-5-113.

[Fel05] Joseph Felsenstein. *PHYLIP (Phylogeny Inference Package) version 3.6. Distributed by the author*. Department of Genome Sciences, University of Washington, Seattle, 2005.

[KM02] L Lacey Knowles and Wayne P Maddison. Statistical phylogeography. *Molecular Ecology*, 11(12):2623–2635, 2002. doi:10.1146/annurev.ecolsys.38.091206.095702.

[LFZK06] Kun Lin, Daniel Yee-Tak Fong, Biliu Zhu, and Johan Karlberg. Environmental factors on the sars epidemic: air temperature, passage of time and multiplicative effect of hospital infection. *Epidemiology & Infection*, 134(2):223–230, 2006. doi:10.1017/S0950268805005054.

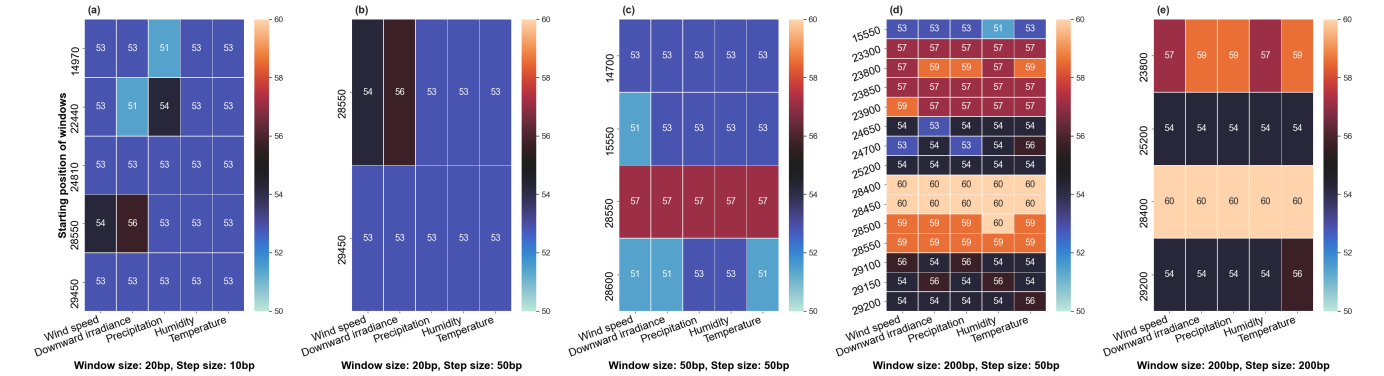


Fig. 3: *Heatmap of Robinson and Foulds topological distance over alignment windows. Five different combinations of parameters were applied (a) window size = 20bp and step size = 10bp; (b) window size = 20bp and step size = 50bp; (c) window size = 50bp and step size = 50bp; (d) window size = 200bp and step size = 50bp; and (e) window size = 200bp and step size = 200bp. Robinson and Foulds topological distance was used to quantify the distance between a phylogenetic tree constructed in certain sliding windows and a climatic tree constructed in corresponding climatic data (wind speed, downward irradiance, precipitation, humidity, temperature).*

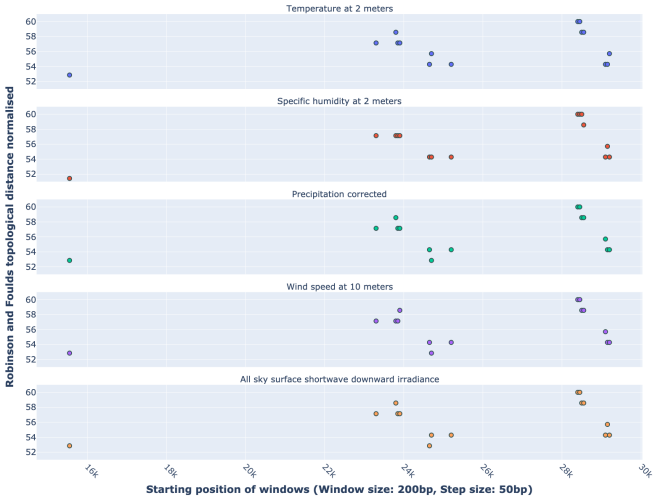


Fig. 4: *Robinson and Foulds topological distance normalized changes over the alignment windows. Multiple phylogenetic analyses were performed using a sliding window (window size = 200 bp and step size = 50 bp). Phylogenetic reconstruction was repeated considering only data within a window that moved along the alignment in steps. The RF normalized topological distance was used to quantify the distance between the phylogenetic tree constructed in each sliding window and the climate tree constructed in the corresponding climate data (Wind speed, Downward irradiance, Precipitation, Humidity, Temperature). Only regions with high genetic mutation rates were marked in the figure.*

[LLKT22] Wanlin Li, My-Lin Luu, Aleksandr Koshkarov, and Nadia Tahiri. aPhylogeo (version 1.0), July 2022. URL: https://github.com/tahiri-lab/aPhylogeo, doi:doi.org/10.5281/zenodo.6773603.

[Nag92] Thomas Nagylaki. Rate of evolution of a quantitative character. *Proceedings of the National Academy of Sciences*, 89(17):8121–8124, 1992. doi:10.1073/pnas.89.17.8121.

[OCFC20] Barbara Oliveiros, Liliana Caramelo, Nuno C Ferreira, and Francisco Caramelo. Role of temperature and humidity in the modulation of the doubling time of covid-19 cases. *MedRxiv*, 2020. doi:10.1101/2020.03.05.20031872.

[OHP+21] Áine O'Toole, Verity Hill, Oliver G Pybus, Alexander Watts, Issac I Bogoch, Kamran Khan, Jane P Messina, The COVID, Genomics UK, et al. Tracking the international spread of sars-cov-2 lineages b. 1.1. 7 and b. 1.351/501y-v2 with grinch. *Wellcome open research*, 6, 2021. doi:10.12688/wellcomeopenres.16661.2.

[OS98] Matthew R Orr and Thomas B Smith. Ecology and speciation. *Trends in Ecology & Evolution*, 13(12):502–506, 1998. doi:10.1016/s0169-5347(98)01511-0.

[OSU+21] Áine O'Toole, Emily Scher, Anthony Underwood, Ben Jackson, Verity Hill, John T McCrone, Rachel Colquhoun, Chris Ruis, Khalil Abu-Dahab, Ben Taylor, et al. Assignment of epidemiological lineages in an emerging pandemic using the pangolin tool. *Virus Evolution*, 7(2):veab064, 2021. doi:10.1093/ve/veab064.

[RF81] David F Robinson and Leslie R Foulds. Comparison of phylogenetic trees. *Mathematical biosciences*, 53(1-2):131–147, 1981. doi:10.1016/0025-5564(81)90043-2.

[RHO+20] Andrew Rambaut, Edward C Holmes, Áine O'Toole, Verity Hill, John T McCrone, Christopher Ruis, Louis du Plessis, and Oliver G Pybus. A dynamic nomenclature proposal for sars-cov-2 lineages to assist genomic epidemiology. *Nature microbiology*, 5(11):1403–1407, 2020. doi:10.1038/s41564-020-0770-5.

[Sch01] Dolph Schluter. Ecology and the origin of species. *Trends in ecology & evolution*, 16(7):372–380, 2001. doi:10.1016/s0169-5347(01)02198-x.

[SDdPS+20] Marcos Felipe Falcão Sobral, Gisleia Benini Duarte, Ana Iza Gomes da Penha Sobral, Marcelo Luiz Monteiro Marinho, and André de Souza Melo. Association between climate variables and global transmission of sars-cov-2. *Science of The Total Environment*, 729:138997, 2020. doi:10.1016/j.scitotenv.2020.138997.

[SMVS+22] Chidambaram Sabarathinam, Prasanna Mohan Viswanathan, Venkatramanan Senapathi, Shankar Karuppannan, Dhanu Radha Samayamanthula, Gnanachandrasamy Gopalakrishnan, Ramanathan Alagappan, and Prosun Bhattacharya. Sars-cov-2 phase i transmission and mutability linked to the interplay of climatic variables: a global observation on the pandemic spread. *Environmental Science and Pollution Research*, pages 1–18, 2022. doi:10.1007/s11356-021-17481-8.

[Sta14] Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014. doi:10.1093/bioinformatics/btu033.

Global optimization software library for research and education

Nadia Udler‡*

Abstract—Machine learning models are often represented by functions given by computer programs. Optimization of such functions is a challenging task because traditional derivative based optimization methods with guaranteed convergence properties cannot be used.. This software allows to create new optimization methods with desired properties, based on basic modules. These basic modules are designed in accordance with approach for constructing global optimization methods based on potential theory [KAP]. These methods do not use derivatives of objective function and as a result work with nondifferentiable functions (or functions given by computer programs, or black box functions), but have guaranteed convergence. The software helps to understand principles of learning algorithms. This software may be used by researchers to design their own variations or hybrids of known heuristic optimization methods. It may be used by students to understand how known heuristic optimization methods work and how certain parameters affect the behavior of the method.

Index Terms—global optimization, black-box functions, algorithmically defined functions, potential functions

Introduction

Optimization lies at the heart of machine learning and data science. One of the most relevant problems in machine learning is automatic selection of the algorithm depending on the objective. This is necessary in many applications such as robotics, simulating biological or chemical processes, trading strategies optimization, to name a few [KHNT]. We developed a library of optimization methods as a first step for self-adapting algorithms. Optimization methods in this library work with all objectives including very onerous ones, such as black box functions and functions given by computer code, and the convergences of methods is guaranteed. This library allows to create customized derivative free learning algorithms with desired properties by combining building blocks from this library or other Python libraries.

The library is intended primarily for educational purposes and its focus is on transparency of the methods rather than on efficiency of implementation.

The library can be used by researches to design optimization methods with desired properties by varying parameters of the general algorithm.

As an example, consider variant of simulated annealing (SA) proposed in [FGSB] where different values of parameters (Boltzman distribution parameters, step size, etc.) are used depending of the distance to optimal point. In this paper the basic SA algorithm is used as a starting point. We can offer more basic module as a starting point (and by specifying distribution as 'exponential' get the variant of SA) thus achieving more flexible design opportunities for custom optimization algorithm. Note that convergence of the newly created hybrid algorithm does not need to be verified when using minpy basic modules, whereas previously mentioned SA-based hybrid has to be verified separately (see [GLUQ])

Testing functions are included in the library. They represent broad range of use cases covering above mentioned difficult functions. In this paper we describe the approach underlying these optimization methods. The distinctive feature of these methods is that they are not heuristic in nature. The algorithms are derived based on potential theory [KAP], and their convergence is guaranteed by their derivation method [KPP]. Recently potential theory was applied to prove convergence of well known heuristic methods, for example see [BIS] for convergence of PSO, and to re prove convergence of well known gradient based methods, in particular, first order methods - see [NBAG] for convergence of gradient descent and [ZALO] for mirror descent. For potential functions approach for stochastic first order optimization methods see [ATFB].

Outline of the approach

The approach works for non-smooth or algorithmically defined functions. For detailed description of the approach see [KAP], [KP]. In this approach the original optimization problem is replaced with a randomized problem, allowing the use of Monte-Carlo methods for calculating integrals. This is especially important if the objective function is given by its values (no analytical formula) and derivatives are not known. The original problem is restated in the framework of gradient (sub gradient) methods, employing the standard theory (convergence theorems for gradient (sub gradient) methods), whereas no derivatives of the objective function are needed. At the same time, the method obtained is a method of nonlocal search unlike other gradient methods. It will be shown, that instead of measuring the gradient of the objective function we can measure the gradient of the potential function at each iteration step , and the value of the gradient can be obtained using values of objective function only, in the framework of Monte Carlo methods for calculating integrals. Furthermore, this value does not have to be precise, because it is recalculated at each iteration step. It will also be shown that well-known zero-order optimization methods (methods that do not use derivatives of objective function but its values only)

* *Corresponding author: nadiakap@optonline.net*
‡ *University of Connecticut (Stamford)*

are generalized into their adaptive extensions. The generalization of zero-order methods (that are heuristic in nature) is obtained using standardized methodology, namely, gradient (sub gradient) framework. We consider the unconstrained optimization problem

$$f(x_1,x_2,..x_n) \to \min_{x\in R_n} \quad (1)$$

By randomizing we get

$$F(X) = E[f(X)] \to \min_{x\in R_n} \quad (2)$$

where X is a random vector from R^n, {X} is a set of such random vectors, and $E[\cdot]$ is the expectation operator.

Problem 2 is equivalent to problem 1 in the sense that any realization of the random vector X^*, where X^* is a solution to 2, that has a nonzero probability, will be a solution to problem 1 (see [KAP] for proof).

Note that 2 is the stochastic optimization problem of the functional F(X) .

To study the gradient nature of the solution algorithms for problem 2, a variation of objective functional F(X) will be considered.

The suggested approach makes it possible to obtain optimization methods in systematic way, similar to the methodology adopted in smooth optimization. Derivation includes randomization of the original optimization problem, finding directional derivative for the randomized problem and choosing moving direction Y based on the condition that directional derivative in the direction of Y is being less or equal to 0.

Because of randomization, the expression for directional derivative doesn't contain the differential characteristics of the original function. We obtain the condition for selecting the direction of search Y in terms of its characteristics - conditional expectation. Conditional expectation is a vector function (or vector field) and can be decomposed (following the theorem of decomposition of the vector field) into the sum of the gradient of scalar function P and a function with zero divergence. P is called a potential function. As a result the original problem is reduced to optimization of the potential function, furthermore, the potential function is specific for each iteration step. Next, we arrive at partial differential equation that connects P and the original function. To define computational algorithms it is necessary to specify the dynamics of the random vectors. For example, the dynamics can be expressed in a form of densities. For certain class of distributions, for example normal distribution, the dynamics can be written in terms of expectation and covariance matrix. It is also possible to express the dynamics in mixed characteristics.

Expression for directional derivative

Derivative of objective functional F(X) in the direction of the random vector Y at the point X^0 (Gateaux derivative) is:

$$\delta_Y F(X^0) = \frac{d}{d\varepsilon}F(X^0+\varepsilon Y)_{\varepsilon=0} = \frac{d}{d\varepsilon}F(X^\varepsilon)dx_{\varepsilon=0} = \frac{d}{d\varepsilon}\int f(X)p_{x^\varepsilon}(x)_{\varepsilon=0}$$

where density function of the random vector $X^\varepsilon = X^0+\varepsilon Y$ may be expressed in terms of joint density function $p_{X^0,Y}(x,y)$ of X^0 and Y as follows:

$$p_{x^\varepsilon}(x) = \int_{R^n} p_{x^\varepsilon}(x-\varepsilon y,y)dy \quad (3)$$

The following relation (property of divergence) will be needed later

$$\frac{d}{d\varepsilon}p_{x^\varepsilon}(x-\varepsilon y,y) = (-\nabla_x p_{x^\varepsilon}(x,y),y) = -div_x(p_{x^\varepsilon}(x,y)y) \quad (4)$$

where (,) defines dot product.

Assuming differentiability of the integrals (for example, by selecting the appropriate $p_{x^\varepsilon}(x,y)$ and using 3, 4 we get

$$\delta_Y F(X^0) = [\frac{d}{d\varepsilon}\int_{R^n}\int_{R^n} f(x)p_{x^\varepsilon}(x-\varepsilon y,y)dxdy]_{\varepsilon=0} =$$

$$= [\tfrac{d}{d\varepsilon}\textstyle\int_{R^n} f(x)\int_{R^n} p_{x^\varepsilon}(x-\varepsilon y,y)dxdy]_{\varepsilon=0} = [\int_{R^n} f(x)(\tfrac{d}{d\varepsilon}\int_{R^n} p_{x^\varepsilon}(x-\varepsilon y,y)dy)dx]_{\varepsilon=0} =$$

$$= \int_{R^n} f(x)(\int_{R^n}[\frac{d}{d\varepsilon}p_{x^\varepsilon}(x-\varepsilon y,y)]_{\varepsilon=0}dy)dx = -\int_{R^n} f(x)(\int_{R^n}[div_x(p_{x^\varepsilon}(x,y)y)]dy)dx =$$

$$-\int_{R^n} f(x)div_x[\int_{R^n}(p_{x^\varepsilon}(x,y)y)dy]dx$$

Using formula for conditional distribution $p_{Y/X^0=x}(y) = \frac{p_{x\varepsilon_y}(x,y)}{p_{x^\varepsilon}(x))}$,

where $p_{x^\varepsilon}(x) = \int_{R^n} p_{x\varepsilon_y}(x,u)du$

we get $\delta_Y F(X^0) = -\int_{R^n} f(x)div_x[p_{x^\varepsilon}(x)\int_{R^n} p_{Y/X^0=x}(y)ydy]dx$

Denote $\bar{y}(x) = \int_{R^n} yp_{Y/X^0=x}(y)dy = E[Y/X^0=x]$

Taking into account normalization condition for density we arrive at the following expression for directional derivative:

$$\delta_Y F(X^0) = -\int_{R^n}(f(x)-C)div_x[p_{x^0}(x)\bar{y}(x)]dx$$

where C is arbitrary chosen constant

Considering solution to $\delta_Y F(X^0) \to \min_Y$ allows to obtain gradient-like algorithms for optimization that use only objective function values (do not use derivatives of objective function)

Potential function as a solution to Poisson's equation

Decomposing vector field $p_{x^0}(x)\bar{y}(x)$ into potential field $\nabla\varphi_0(x)$ and divergence-free component $W_0(x)$:

$$p_{x^0}(x)\bar{y}(x) = \nabla\phi_0(x) + W_0(x)$$

we arrive at Poisson's equation for potential function:

$$\Delta\varphi_0(x) = -L[f(x)-C]p_u(x)$$

where L is a constant

Solution to Poisson's equation approaching 0 at infinity may be written in the following form

$$\varphi_0(x) = \int_{R^n} E(x,\xi)[f(\xi)-C]p_u(\xi)d\xi$$

where $E(x,\xi)$ is a fundamental solution to Laplace's equation.

Then for potential component $\Delta\varphi_0(x)$ we have

$$\Delta\varphi_0(x) = -LE[\Delta_x E(x,u)(f(x)-C)]$$

To conclude, the representation for gradient-like direction is obtained. This direction maximizes directional derivative of the objective functional F(X). Therefore, this representation can be used for computing the gradient of the objective function f(x) using only its values. Gradient direction of the objective function f(x) is determined by the gradient of the potential function $\varphi_0(x)$, which, in turn, is determined by Poisson's equation.

Practical considerations

The dynamics of the expectation of objective function may be written in the space of random vectors as follows:

$$X_{N+1} = X_N + \alpha_{N+1} Y_{N+1}$$

where N - iteration number, Y^{N+1} - random vector that defines direction of move at (N+1)th iteration, α_{N+1} -step size on (N+1)th iteration. Y^{N+1} must be feasible at each iteration, i.e. the objective functional should decrease: $F(X^{N+1}) < (X^N)$. Applying expection to (12) and presenting $E[Y_{N+1}$ asconditional expectation $E_x E[Y/X]$ we get:

$$X_{N+1} = E[X_N] + \alpha_{N+1} E_{X^N} E[Y^{N+1}/X^N]$$

Replacing mathematical expectations $E[X_N]$ and $Y_{N+1}]$ with their estimates $\overline{E}^{N+1}$ and $\overline{y}(X^N)$ we get:

$$\overline{E}^{N+1} = \overline{E}^N + \alpha_{N+1} \overline{E}_{X^N}[\overline{y}(X^N)]$$

Note that expression for $\overline{y}(X^N)$ was obtained in the previos section up to certain parameters. By setting parameters to certain values we can obtain stochastic extensions of well known heuristics such as Nelder and Mead algorithm or Covariance Matrix Adaptation Evolution Strategy. In minpy library we use several common building blocks to create different algorithms. Customized algorithms may be defined by combining these common blocks and varying their parameters.

Main building blocks include computing center of mass of the sample points and finding newtonian potential.

Key takeaways, example algorithm, and code organization

Many industry professionals and researchers utilize mathematical optimization packages to search for better solutions of their problems. Examples of such problem include minimization of free energy in physical system [FW], robot gait optimization from robotics [PHS], designing materials for 3D printing [ZM], [TMAACBA], wine production [CTC], [CWC], optimizing chemical reactions [VNJT]. These problems may involve "black box optimization", where the structure of the objective function is unknown and is revealed through a small sequence of expensive trials. Software implementations for these methods become more user friendly. As a rule, however, certain modeling skills are needed to formulate real world problem in a way suitable for applying software package. Moreover, selecting optimization method appropriate for the model is a challenging task. Our educational software helps users of such optimization packages and may be considered as a companion to them. The focus of our software is on transparency of the methods rather than on efficiency. A principal benefit of our software is the unified approach for constructing algorithms whereby any other algorithm is obtained from the generalized algorithm by changing certain parameters. Well known heuristic algorithms such as Nelder and Mead (NM) algorithm may be obtained using this generalized approach, as well as new algorithms. Although some derivative-free optimization packages (matlab global optimization toolbox, Tensorflow Probability optimizers, Excel Evolutionary Solver, scikit-learn Stochastic Gradient Descent class, scipy.optimize.shgo method) put a lot of effort in transparency and educational value, they don't have the same level of flexibility and generality as our system. An example of educational-only optimization software is [SAS]. It is limited to teach Particle Swarm Optimization.

The code is organized in such a way that it allows to pair the algorithm with objective function. The new algorithm may be implmented as method of class Minimize. Newly created algorithm can be paired with test objectivve function supplied with a library or with externally supplied objective function (implemented in separate python module). New algorithms can be made more or less universal, that is, may have different number of parameters that user can specify. For example, it is possible to create Nelder and Mead algorithm (NM) using basic modules, and this would be an example of the most specific algorithm. It is also possible to create Stochastic Extention of NM (more generic than classic NM, similar to Simplicial Homology Global Optimisation [ESF] method) and with certain settings of adjustable parameters it may work identical to classic NM. Library repository may be found here: https://github.com/nadiakap/MinPy_edu

The following algorithms demonstrate steps similar to steps of Nelder and Mead algorithm (NM) but select only those points with objective function values smaller or equal to mean level of objective funtion. Such an improvement to NM assures its convergence [KPP]. Unlike NM, they are derived from the generic approach. First variant (NM-stochastic) resembles NM but corrects some of its drawbacks, and second variant (NM-nonlocal) has some similarity to random search as well as to NM and helps to resolve some other issues of classical NM algorithm.

Steps of NM-stochastic:

1) Initialize the search by generating $K \geq n$ separate realizations of u_0^i, i=1,..K of the random vector U_0, and set $m_0 = \frac{1}{K}\sum_{i=0}^{K} u_0^i$
2) On step j = 1, 2, ...

a.Compute the mean level $c_{j-1} = \frac{1}{K}\sum_{i=1}^{K} f(u_{j-1}^i)$

b.Calculate new set of vertices:

$$u_j^i = m_{j-1} + \varepsilon_{j-1}(f(u_{j-1}^i) - c_{j-1})\frac{m_{j-1} - u_{j-1}^i}{||m_{j-1} - u_{j-1}^i||^n}$$

c.Set $m_j = \frac{1}{K}\sum_{i=0}^{K} u_j^i$

d.Adjust the step size ε_{j-1} so that $f(m_j) < f(m_{j-1})$. If approximate ε_{j-1} cannot be obtained within the specified number of trails, then set $m_k = m_{j-1}$

e.Use sample standard deviation as termination criterion:

$$D_j = (\frac{1}{K-1}\sum_{i=1}^{K}(f(u_j^i) - c_j)^2)^{1/2}$$

Note that classic simplex search methods do not use values of objective function to calculate reflection/expantion/contraction coefficients. Those coefficients are the same for all vertices, whereas in NM-stochastic the distance each vertex will travel depends on the difference between objective function value and average value across all vertices $(f(u_j^i) - c_j)$. NM-stochastic shares the following drawbacks with classic simplex methods: a. simlex may collapse into a nearly degenerate figure, and usually proposed remedy is to restart the simlex every once in a while, b. only initial vertices are randomly generated, and the path of all subsequent vertices is deterministic. Next variant of the algorithm (NM-nonlocal) maintains the randomness of vertices on each step, while adjusting the distribution of U_0 to mimic the pattern of the modified vertices. The corrected algorithm has much higher exploration power than the first algorithm (similar to the exploration power of random search algorithms), and has exploitation power of direct - search algorithms.

Steps of NM - nonlocal

1) Choose a starting point x_0 and set $m_0 = x_0$.

2. On step j = 1, 2, ... Obtain K separate realizations of u_i^i, i=1,..K of the random vector U_j

a.Compute $f(u_{j-1}^i), j = 1, 2, ..K$, and the sample mean level

$$c_{j-1} = \frac{1}{K}\sum_{i=1}^{K} f(u_{j-1}^i)$$

b.Generate the new estimate of the mean:

$$m_j = m_{j-1} + \varepsilon_j \frac{1}{K}\sum_{i=1}^{K}[(f(u_j^i) - c_j)\frac{m_{j-1} - u_j^i}{||m_{j-1} - u_j^i||^n}]$$

Adjust the step size ε_{j-1} so that $f(m_j) < f(m_{j-1})$. If approximate ε_{j-1} cannot be obtained within the specified number of trails, then set $m_k = m_{j-1}$

c.Use sample standard deviation as termination criterion

$$D_j = (\frac{1}{K-1}\sum_{i=1}^{K}(f(u_j^i) - c_j)^2)^{1/2}$$

References

[KAP] Kaplinskii, A.I.,Pesin, A.M.,Propoi, A.I.(1994). Analysis of search methods of optimization based on potential theory. I: Nonlocal properties. Automation and Remote Control. Volume 55, N.9, Part 2, September, pp.1316-1323 (rus. pp.97-105), 1994

[KP] Kaplinskii, A.I. and Propoi, A.I., Nonlocal Optimization Methods ofthe First Order Based on Potential Theory, Automation and Remote Control. Volume 55, N.7, Part 2, July, pp.1004-1011 (rus. pp.97-102), 1994

[KPP] Kaplinskii, A.I., Pesin, A.M.,Propoi, A.I. Analysis of search methods of optimization based on potential theory. III: Convergence of methods. Automation and remote Control, Volume 55, N.11, Part 1, November, pp.1604-1610 (rus. pp.66-72), 1994.

[NBAG] Nikhil Bansal, Anupam Gupta, Potential-function proofs for gradient methods, Theory of Computing, Volume 15, (2019) Article 4 pp. 1-32, https://doi.org/10.4086/toc.2019.v015a004

[ATFB] Adrien Taylor, Francis Bach, Stochastic first-order methods: non-asymptotic and computer-aided analyses via potential functions, arXiv:1902.00947 [math.OC], 2019, https://doi.org/10.48550/arXiv.1902.00947

[ZALO] Zeyuan Allen-Zhu and Lorenzo Orecchia, Linear Coupling: An Ultimate Unification of Gradient and Mirror Descent, Innovations in Theoretical Computer Science Conference (ITCS), 2017, pp. 3:1-3:22, https://doi.org/10.4230/LIPIcs.ITCS.2017.3

[BIS] Berthold Immanuel Schmitt, Convergence Analysis for Particle Swarm Optimization, FAU University Press, 2015

[FGSB] FJuan Frausto-Solis, Ernesto Liñán-García, Juan Paulo Sánchez-Hernández, J. Javier González-Barbosa, Carlos González-Flores, Guadalupe Castilla-Valdez, Multiphase Simulated Annealing Based on Boltzmann and Bose-Einstein Distribution Applied to Protein Folding Problem, Advances in Bioinformatics, Volume 2016, Article ID 7357123, https://doi.org/10.1155/2016/7357123

[GLUQ] Gong G., Liu, Y., Qian M, Simulated annealing with a potential function with discontinuous gradient on R^d, Ici. China Ser. A-Math. 44, 571-578, 2001, https://doi.org/10.1007/BF02876705

[PHS] Valdez, S.I., Hernandez, E., Keshtkar, S. (2020). A Hybrid EDA/Nelder-Mead for Concurrent Robot Optimization. In: Madureira, A., Abraham, A., Gandhi, N., Varela, M. (eds) Hybrid Intelligent Systems. HIS 2018. Advances in Intelligent Systems and Computing, vol 923. Springer, Cham. https://doi.org/10.1007/978-3-030-14347-3_20

[FW] Fan, Yi & Wang, Pengjun & Heidari, Ali Asghar & Chen, Huiling & HamzaTurabieh, & Mafarja, Majdi, 2022. "Random reselection particle swarm optimization for optimal design of solar photovoltaic modules," Energy, Elsevier, vol. 239(PA), https://doi.org/10.1016/j.energy.2021.121865

[VNJT] Fath, Verena, Kockmann, Norbert, Otto, Jürgen, Röder, Thorsten, Self-optimising processes and real-time-optimisation of organic syntheses in a microreactor system using Nelder–Mead and design of experiments, React. Chem. Eng., 2020,5, 1281-1299, https://doi.org/10.1039/D0RE00081G

[ZM] Plüss, T.; Zimmer, F.; Hehn, T.; Murk, A. Characterisation and Comparison of Material Parameters of 3D-Printable Absorbing Materials. Materials 2022, 15, 1503. https://doi.org/10.3390/ma15041503

[TMAACBA] Thoufeili Taufek, Yupiter H.P. Manurung, Mohd Shahriman Adenan, Syidatul Akma, Hui Leng Choo, Borhen Louhichi, Martin Bednardz, and Izhar Aziz.3D Printing and Additive Manufacturing, 2022, http://doi.org/10.1089/3dp.2021.0197

[CTC] Vismara, P., Coletta, R. & Trombettoni, G. Constrained global optimization for wine blending. Constraints 21, 597–615 (2016), https://doi.org/10.1007/s10601-015-9235-5

[CWC] Terry Hui-Ye Chiu, Chienwen Wu, Chun-Hao Chen, A Generalized Wine Quality Prediction Framework by Evolutionary Algorithms, International Journal of Interactive Multimedia and Artificial Intelligence, Vol. 6, Nº7,2021, https://doi.org/10.9781/ijimai.2021.04.006

[KHNT] Pascal Kerschke, Holger H. Hoos, Frank Neumann, Heike Trautmann; Automated Algorithm Selection: Survey and Perspectives. Evol Comput 2019; 27 (1): 3–45, https://doi.org/10.1162/evco_a_00242

[SAS] Leandro dos Santos Coelho, Cezar Augusto Sierakowski, A software tool for teaching of particle swarm optimization fundamentals, Advances in Engineering Software, Volume 39, Issue 11, 2008, Pages 877-887, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2008.01.005.

[ESF] Endres, S.C., Sandrock, C. & Focke, W.W. A simplicial homology algorithm for Lipschitz optimisation. J Glob Optim 72, 181–217 (2018), https://doi.org/10.1007/s10898-018-0645-y

Temporal Word Embeddings Analysis for Disease Prevention

Nathan Jacobi[‡*], Ivan Mo[‡§], Albert You[‡], Krishi Kishore[‡], Zane Page[‡], Shannon P. Quinn[‡¶], Tim Heckman[‖]

Abstract—Human languages' semantics and structure constantly change over time through mediums such as culturally significant events. By viewing the semantic changes of words during notable events, contexts of existing and novel words can be predicted for similar, current events. By studying the initial outbreak of a disease and the associated semantic shifts of select words, we hope to be able to spot social media trends to prevent future outbreaks faster than traditional methods. To explore this idea, we generate a temporal word embedding model that allows us to study word semantics evolving over time. Using these temporal word embeddings, we use machine learning models to predict words associated with the disease outbreak.

Index Terms—Natural Language Processing, Word Embeddings, Bioinformatics, Social Media, Disease Prediction

Introduction & Background

Human languages experience continual changes to their semantic structures. Natural language processing techniques allow us to examine these semantic alterations through methods such as word embeddings. Word embeddings provide low dimension numerical representations of words, mapping lexical meanings into a vector space. Words that lie close together in this vector space represent close semantic similarities [MCCD13]. This numerical vector space allows for quantitative analysis of semantics and contextual meanings, allowing for more use in machine learning models that utilize human language.

We hypothesize that disease outbreaks can be predicted faster than traditional methods by studying word embeddings and their semantic shifts during past outbreaks. By surveying the context of select medical terms and other words associated with a disease during the initial outbreak, we create a generalized model that can be used to catch future similar outbreaks quickly. By leveraging social media activity, we predict similar semantic trends can be found in real time. Additionally, this allows novel terms to be evaluated in context without requiring a priori knowledge of them, allowing potential outbreaks to be detected early in their lifespans, thus minimizing the resultant damage to public health.

Given a corpus spanning a fixed time period, multiple word embeddings can be created at set temporal intervals, which can then be studied to track contextual drift over time. However, a common issue in these so-called "temporal word embeddings" is that they are often unaligned — i.e. the embeddings do not lie within the same embedding space. Past proposed solutions to aligning temporal word embeddings require multiple separate alignment problems to be solved, or for "anchor words" – words that have no contextual shifts between times – to be used for mapping one time period to the next [HLJ16]. Yao et al. propose a solution to this alignment issue, shown to produce accurate and aligned temporal word embeddings, through solving one joint alignment problem across all time slices, which we utilize here [YSD+18].

* *Corresponding author: Nathan.Jacobi@uga.edu*
‡ *Computer Science Department, University of Georgia*
§ *Linguistics Department, University of Georgia*
¶ *Cellular Biology Department, University of Georgia*
‖ *Public Health Department, University of Georgia*

Methodology

Data Collection & Pre-Processing

Our data set is a corpus D of over 7 million tweets collected from Scott County, Indiana from the dates January 1st, 2014 until January 17th, 2017. The data was lent to us from Twitter after a data request, and has not yet been made publicly available. During this time period, an HIV outbreak was taking place in Scott County, with an eventual 215 confirmed cases being linked to the outbreak [PPH+16]. Gonsalves et al. predicts an additional 126 undiagnosed HIV cases were linked to this same outbreak [GC18]. The state's response led to questioning if the outbreak could have been stemmed or further prevented with an earlier response [Gol17]. Our corpus was selected with a focus on tweets related to the outbreak. By closely studying the semantic shifts during this outbreak, we hope to accurately predict similar future outbreaks before they reach large case numbers, allowing for a critical earlier response.

To study semantic shifts through time, the corpus was split into 18 temporal buckets, each spanning a 2 month period. All data utilized in scripts was handled via the pandas Python package. The corpus within each bucket is represented by D_t, with t representing the temporal slice. Within each 2 month period, tweets were split into 12 pre-processed output csv files. Pre-processing steps first removed retweets, links, images, emojis, and punctuation. Common stop words were removed from the tweets using the NLTK Python package, and each tweet was tokenized. A vocabulary dictionary was then generated for each of the 18 temporal buckets, containing each unique word and a count of its occurrences within its respective bucket. The vocabulary dictionaries for each bucket were then combined into a global vocabulary dictionary, containing the total counts for each unique word across all 18 buckets. Our experiments utilized two vocabulary dictionaries: the

first being the 10,000 most frequently occurring words from the global vocabulary for ensuring proper generation of embedding vectors, the second being a combined vocabulary of 15,000 terms, including our target HIV/AIDS related terms. This combined vocabulary consisted of the top 10,000 words across D as well as an additional 473 HIV/AIDS related terms that occurred at least 8 times within the corpus. The 10,000th most frequent term in D occurred 39 times, so to ensure results were not influenced by sparsity in the less frequent HIV/AIDS terms, 4,527 randomly selected terms with occurrences between 10 and 25 times were added to the vocabulary, bringing it to a total of 15,000 terms. The HIV/AIDS related terms came from a list of 1,031 terms we compiled, primarily coming from the U.S. Department of Veteran Affairs published list of HIV/AIDS related terms, and other terms we thought were pertinent to include, such as HIV medications and terms relating to sexual health [Aff05].

Temporally Aligned Vector Generation

Generating word2vec embeddings is typically done through 2 primary methods: continuous bag-of-words (CBOW) and skip-gram, however many other various models exist [MCCD13]. Our methods use a CBOW approach at generating embeddings, which generates a word's vector embedding based on the context the word appears in, i.e. the words in a window range surrounding the target word. Following pre-processing of our corpus, steps for generating word embeddings were applied to each temporal bucket. For each time bucket, co-occurrence matrices were first created, with a window size w = 5. These matrices contained the total occurrences of each word against every other within a window range L of 5 words within the corpus at time t. Each co-occurrence matrix was of dimensions $|V| \times |V|$. Following the generation of each of these co-occurrence matrices, a $|V| \times |V|$ dimensioned Positive Pointwise Mutual Information matrix was calculated. The value in each cell was calculated as follows:

$$\text{PPMI}(t,L)_{w,c} = \max\{\text{PMI}(D_t,L)_{w,c}, 0\},$$

where w and c are two words in V. Embeddings generated by word2vec can be approximated by PMI matrices, where given embedding vectors utilize the following equation [YSD+18]:

$$u_w^T u_c \approx \text{PMI}(D,L)_{w,c}$$

Each embedding u has a reduced dimensionality d, typically around 25 - 200. Each PPMI from our data set is created independently from each other temporal bucket. After these PPMI matrices are made, temporal word embeddings can be created using the method proposed by Yao et al. [YSD+18]. The proposed solution focuses on the equation:

$$U(t)U(t)^T \approx \text{PPMI}(t,L)$$

where U is a set of embeddings from time period t. Decomposing each PPMI(t) will yield embedding $U(t)$, however each $U(t)$ is not guaranteed to be in the same embedding space. Yao et al. derives $U(t)A = B$ with the following equation[2][3][4] [YSD+18]:

$$A = U(t)^T U(t) + (\gamma + \lambda + 2\tau)I,$$

$$B = Y(t)U(t) + \gamma U(t) + \tau(U(t-1) + U(t+1))$$

To decompose PPMI(t) in our model, SciPy's linear algebra package was utilized to solve for eigendecomposition of each PPMI(t), and the top 100 terms were kept to generate an embedding of d = 100. The alignment was then applied, yielding 18 temporally aligned word embedding sets of our vocabulary, with dimensions $|V| \times d$, or 15,000 x 100. These word embedding sets are aligned spatially and in terms of rotations, however there appears to be some spatial drift that we hope to remove by tuning hyperparameters. Following alignment, these vectors are usable for experimentation and analysis.

Predictions for Detecting Modern Shifts

Following the generation of temporally aligned word embedding, they can be used for semantic shift analysis. Using the word embedding vectors generated for each temporal bucket, 2 new data sets were created to use for determining patterns in the semantic shifts surrounding HIV outbreaks. Both of these data sets were constructed using our second vocabulary of 15,000 terms, including the 473 HIV/AIDS related terms, and each term's embedding of d = 100 that were generated by the dynamic embedding model. The first experimental data set was the shift in the d = 100 embedding vector between each time bucket and the one that immediately followed it. These shifts were calculated by simply subtracting the next temporal and initial vectors from each other. In addition to the change in the 100 dimensional vector between each time bucket and its next, the initial and next 10 dimensional embeddings were included from each, which were generated using the same dynamic embedding model. This yielded each word having 17 observations and 121 features: {d_vec0 ... d_vec99, v_init_0 ... v_init_9, v_fin_0 ... v_fin_9, label}. This data set will be referred to as "data_121". The reasoning to include these lower dimensional embeddings was so that both the shift and initial and next positions in the embedding space would be used in our machine learning algorithms. The other experimental data set was constructed similarly, but rather than subtracting the two vectors and including lower dimensions vectors, the initial and next 100 dimensional vectors were listed as features. This allowed machine learning algorithms to have access to the full positional information of each vector alongside the shift between the two. This yielded each word having 17 observations and 201 features: {vec_init0 ... vec_init99, vec_fin0 ... vec_fin99, label}. This data set will be referred to as "data_201". With the 15,000 terms each having 17 observations, it led to a total of 255,000 observations. It should be noted that in addition to the vector information, the data sets also listed the number of days since the outbreak began, the predicted number of cases at that point in time, from [GC18], and the total magnitude of the shift in the vector between the corresponding time buckets. All these features were dropped prior to use within the models, as the magnitude feature was colinear with the other positional features, and the case and day data will not be available in predicting modern outbreaks. Using these data, two machine learning algorithms were applied: unsupervised k-means clustering and a supervised neural network.

K-means Clustering

To examine any similarities within shifts, k-means clustering was performed on the data sets at first. Initial attempts at k-means with the 100 dimensional embeddings yielded extremely large inertial values and poor results. In an attempt to reduce inertia, features

1. All code used can be found here https://github.com/quinngroup/Twitter-Embedding-Analysis/

2. γ represents the forcing regularizer. λ represents the Frobenius norm regularizer. τ represents the smoothing regularizer.

3. Y(t) represents PPMI(t).

4. The original equation uses W(t), but this acts as identical to U(t) in the code. We replaced it here to improve readability.

for data that k-means would be performed onto were assessed. K-means was performed on a reduced dimensionality data set, with embedding vectors of dimensionality d = 10, however this led to strict convergence and poor results again. The data set with the change in an embeddings vector, data_121, continued to contain the changes of vectors between each time bucket and its next. However, rather than the 10 dimensional position vectors for both time buckets, 2 dimensional positions were used instead, generated by UMAP from the 10 dimensioned vectors. The second data set, data_201, always led to strict convergence on clustering, even when reduced to just the 10 dimensional representations. Therefore, k-means was performed explicitly on the data_121 set, with the 2 dimensional representations alongside the 100 dimensional change in the vectors. Separate two dimensional UMAP representations were generated for use as a feature and for visual examination. The data set also did not have the term's label listed as a feature for clustering.

Inertia at convergence on clustering for k-means was reduced significantly, as much as 86% after features were reassessed, yielding significantly better results. Following the clustering, the results were analyzed to determine which clusters contained the higher than average incidence rates of medical terms and HIV/AIDS related terms. These clusters can then be considered target clusters, and large incidences of words being clustered within these can be flagged as indicative as a possible outbreak.

Neural Network Predictions

In addition to the k-means model, we created a neural network model for binary classification of our terms. Our target class was terms that we hypothesized were closely related to the HIV epidemic in Scott County, i.e. any word in our HIV terms list. Several iterations with varying number of layers, activation functions, and nodes within each layer were attempted to maximize performance. Each model used an 80% training, 20% testing split on these data, with two variations performed of this split on training and testing data. The first was randomly splitting all 255,000 observations, without care of some observations for a term being in both training set and some being in the testing set. This split of data will be referred to as "mixed" data, as the terms are mixed between the splits. The second split of data split the 15,000 words into 80% training and 20% testing. After the vocabulary was split, the corresponding observations in the data were split accordingly, leaving all observations for each term within the same split. Additionally, we tested a neural network that would accept the same data as the input, either data_201 or data_121, with the addition of the label assigned to that observation by the k-means model as a feature. The goal of these models, in addition was to correctly identifying terms we classified as related to the outbreak, was to discover new terms that shift in similar ways to the HIV terms we labeled.

The neural network model used was four layers, with three ReLu layers with 128, 256, and 256 neurons, followed by a single neuron sigmoid output layer. This neural network was constructed using the Keras module of the TensorFlow library. The main difference between them was the input data itself. The input data were data_201 with and without k-means labels, data_121 with and without k-means labels. On each of these, there were two splits of the training and testing data, as in the previously mentioned "mixed" terms. Parameters of the neural network layers were adjusted, but results did not improve significantly across the data sets. All models were trained with a varying number of epochs: 50, 100, 150, and 200. Additionally, several certainty thresholds for a positive classification were tested on each of the models. The best results from each will be listed in the results section. As we begin implementation of these models on other HIV outbreak related data sets, the proper certainty thresholds can be better determined.

Results

Analysis of Embeddings

To ensure accuracy in word embeddings generated in this model, we utilized word2vec (w2v), a proven neural network method of embeddings [MCCD13]. For each temporal bucket, a static w2v embedding of d = 100 was generated to compare to the temporal embedding generated from the same bucket. These vectors were generated from the same corpus as the ones generated by the dynamic model. As the vectors do not lie within the same embedding space, the vectors cannot be directly compared. As the temporal embeddings generated by the alignment model are influenced by other temporal buckets, we hypothesize notably different vectors. Methods for testing quality in [YSD+18] rely on a semi-supervised approach: the corpus used is an annotated set of New York Times articles, and the section (Sports, Business, Politics, etc.) are given alongside the text, and can be used to assess strength of an embedding. Additionally, the corpus used spans over 20 years, allowing for metrics such as checking the closest word to leaders or titles, such as "president" or "NYC mayor" throughout time. These methods show that this dynamic word embedding alignment model yields accurate results.

Major differences can be attributed to the word2vec model only being given a section of the corpus at a time, while our model had access to the entire corpus across all temporal buckets. Terms that might not have appeared in the given time bucket might still appear in the embeddings generated by our model, but not at all within the word2vec embeddings. For example, most embeddings generated by the word2vec model did not often have hashtagged terms in their top 10 closest terms, while embeddings generated by our model often did. As hashtagged terms are very related to ongoing events, keeping these terms can give useful information to this outbreak. Modern hashtagged terms will likely be the most common novel terms that we have no prior knowledge on, and we hypothesize that these terms will be relevant to ongoing outbreaks.

Given that our corpus spans a significantly shorter time period than the New York Times set, and does not have annotations, we use existing baseline data sets of word similarities. We evaluated the accuracy of both models' vectors using a baseline sources for the semantic similarity of terms. The first source used was SimLex-999, which contains 999 word pairings, with corresponding human generated similarity scores on a scale of 0-10, where 10 is the highest similarity [HRK15]. Cosine similarities for each pair of terms in SimLex-999 were calculated for both the w2v model vectors as well as vectors generated by the dynamic model for each temporal bucket. Pairs containing terms that were not present in the model generated vectors were omitted for that models similarity measurements. The cosine similarities were then compared to the assigned SimLex scores using the Spearman's rank correlation coefficient. The results of this baseline can be seen in Table 1. The Spearman's coefficient of both sets of embeddings, averaged across all 18 temporal buckets, was .151334 for the w2v vectors and .15506 for the dynamic word embedding (dwe) vectors. The dwe vectors slightly outperformed the w2v baseline in this test of word similarities. However, it should be noted that

Time Bucket	w2v Score (MEN)	dwe Score (MEN)	Difference (MEN)	w2v Score (SL)	dwe Score (SL)	Difference (SL)
0	0.437816	0.567757	0.129941	0.136146	0.169702	0.033556
1	0.421271	0.561996	0.140724	0.131751	0.167809	0.036058
2	0.481644	0.554162	0.072518	0.113067	0.165794	0.052727
3	0.449981	0.543395	0.093413	0.137704	0.163349	0.025645
4	0.360462	0.532634	0.172172	0.169419	0.158774	-0.010645
5	0.353343	0.521376	0.168032	0.133773	0.157173	0.023400
6	0.365653	0.511323	0.145669	0.173503	0.154299	-0.019204
7	0.358100	0.502065	0.143965	0.196332	0.152701	-0.043631
8	0.380266	0.497222	0.116955	0.152287	0.154338	.002051
9	0.405048	0.496563	0.091514	0.149980	0.148919	-0.001061
10	0.403719	0.499463	0.095744	0.145412	0.142114	-0.003298
11	0.381033	0.504986	0.123952	0.181667	0.141901	-0.039766
12	0.378455	0.511041	0.132586	0.159254	0.144187	-0.015067
13	0.391209	0.514521	0.123312	0.145519	0.147816	0.002297
14	0.405100	0.519095	0.113995	0.151422	0.152477	0.001055
15	0.419895	0.522854	0.102959	0.117026	0.154963	0.037937
16	0.400947	0.524462	0.123515	0.158833	0.157687	-0.001146
17	0.321936	0.525109	0.203172	0.170925	0.157068	-0.013857
Average	**0.437816**	**0.567757**	**0.129941**	**0.151334**	**0.155059**	**0.003725**

TABLE 1: *Spearman's correlation coefficients for w2v vectors and dynamic word embedding (dwe) vectors for all 18 temporal clusters against the SimLex word pair data set.*

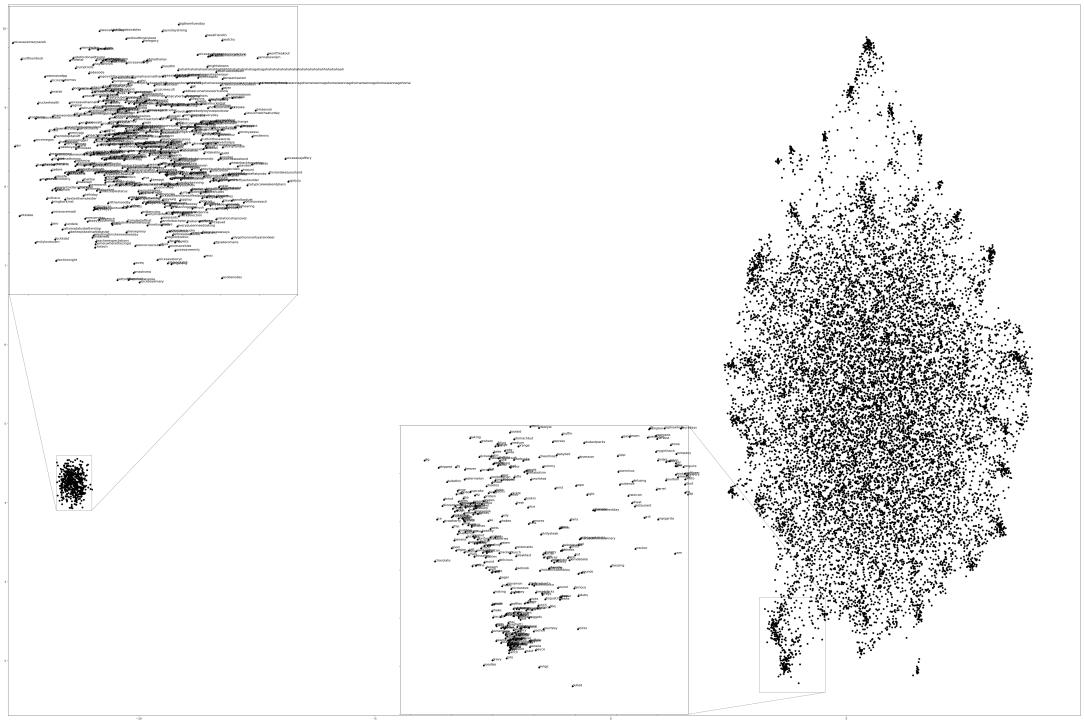


Fig. 1: *2 Dimensional Representation of Embeddings from Time Bucket 0.*

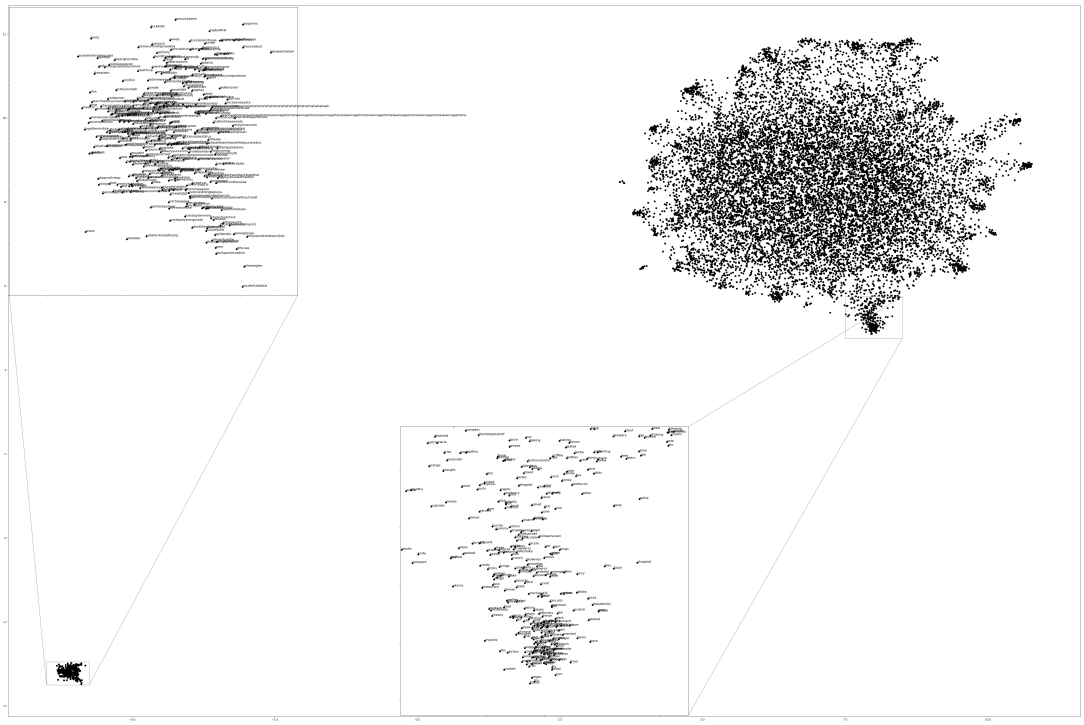


Fig. 2: *2 Dimensional Representation of Embeddings from Time Bucket 17.*

these Spearman's coefficients are very low compared to baselines such as in [WWC+19], where the average Spearman's coefficient amongst common models was .38133 on this data set of words. These models, however, were trained on corpus generated from Wikipedia pages — wiki2010. The lower Spearman's coefficients can likely be accounted to our corpus. In 2014-2017, when this corpus was generated, Twitter had a 140 character limit on tweets. The limited characters have been shown to affect user's language within their tweets [BTKSDZ19], possibly affecting our embeddings. Boot et al. show that Twitter increasing the character limit to 280 characters in 2017 impacted the language within the tweets. As we test this pipeline on more Twitter data from various time intervals, the character increase in 2017 is something to keep in mind.

The second source of baseline was the MEN Test Collection, containing 3,000 pairs with similarity scores of 0-50, with 50 being the most similar [BTB14]. Following the same methodology for assessing the strength of embeddings as we did for the SimLex-999 set, the Spearman's coefficients from this set yielded much better results than from the SimLex-999 set. The average of the Spearman's coefficients, across all 18 temporal buckets, was .39532 for the w2v embeddings and .52278 for the dwe embeddings. The dwe significantly outperformed the w2v baseline on this set, but still did not reach the average correlation of .7306 that other common models achieved in the baseline tests in [WWC+19].

Two dimensional representations of embeddings, generated by UMAP, can be seen in Figure 1 and Figure 2. Figure 1 represents the embedding generated for the first time bucket, while Figure 2 represents the embedding generated for the final time bucket. These UMAP representations use cosine distance as their metric over Euclidian distance, leading to more dense clusters and more accurate representations of nearby terms within the embedding space. The section of terms outlying from the main grouping appears to be terms that do not appear often within that temporal cluster itself, but may appear several times later in a temporal bucket. Figure 1 contains a zoomed in view of this outlying group, as well as a subgrouping on the outskirts of the main group, containing food related terms. The majority of these terms are ones that would likely be hashtagged frequently during a brief time period within one temporal bucket. These terms are still relevant to study, as hashtagged terms that appear frequently for a brief period of time are most likely extremely attached to an ongoing event. In future iterations, the length of each temporal bucket will be decreased, hopefully giving more temporal buckets access to terms that only appear within one currently.

K-Means Clustering Results

The results of the k-means clustering can be seen below in Figures 4 and 5. Figure 4 shows the results of k-means clustering with the corresponding 2 dimensional UMAP positions generated from the 10 dimensional vector that were used as features in the clustering. Figure 5 shows the results of k-means clustering with the corresponding 2 dimensional UMAP representation of the

Cluster	All Words	HIV Terms	Difference
0	0.173498	0.287048	0.113549
1	0.231063	0.238876	0.007814
2	0.220039	0.205600	-0.014440
3	0.023933	0.000283	-0.023651
4	0.108078	0.105581	-0.002498
5	0.096149	0.084276	-0.011873
6	0.023525	0.031391	0.007866
7	0.123714	0.046946	-0.076768

TABLE 2: *Distribution of HIV terms and all terms within k-means clusters*

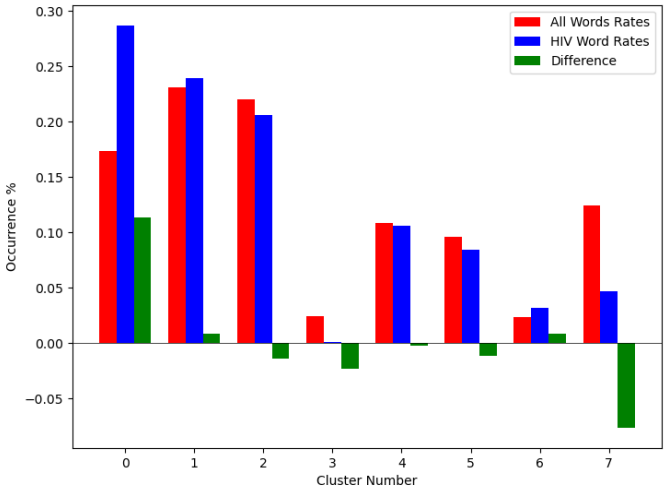


Fig. 3: *Bar graph showing k-means clustering distribution of HIV terms against all terms.*

entire data set used in clustering. The k-means clustering revealed semantic shifts of HIV related terms being clustered with higher incidence than other terms in one cluster. Incidence rates for all terms and HIV terms in each cluster can be seen in Table 2 and Figure 3. This increased incidence rate of HIV related terms in certain clusters leads us to hypothesize that semantic shifts of terms in future datasets can be clustered using the same k-means model, and analyzed to search for outbreaks. Clustering of terms in future data sets can be compared to these clustering results, and similarities between the data can be recognized.

Neural Network Results

Neural network models we generated showed promising results on classification of HIV related terms. The goal of the models was to identify and discover terms surrounding the HIV outbreak. Therefore we were not concerned about the rate of false positive terms. False positive terms likely had semantic shifts very similar to the HIV related terms, and therefore can be related to the outbreak. These terms can be labeled as potentially HIV related while studying future data sets, which can aid the identifying of if an outbreak is ongoing during the time tweets in the corpus were tweeted. We looked for a balance of finding false positive terms without lowering our certainty threshold to include too many terms. Results of the testing data for data_201 set can be seen in 3, and results of the testing data for data_121 set can be seen in 4. The certainty threshold for the unmixed split in both sets was .01,

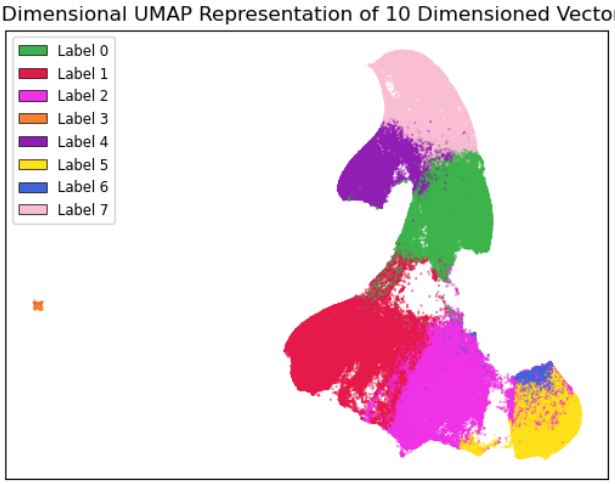


Fig. 4: *Results of k-means clustering shown over the 2 dimensional UMAP representation of the 10 dimensional embeddings.*

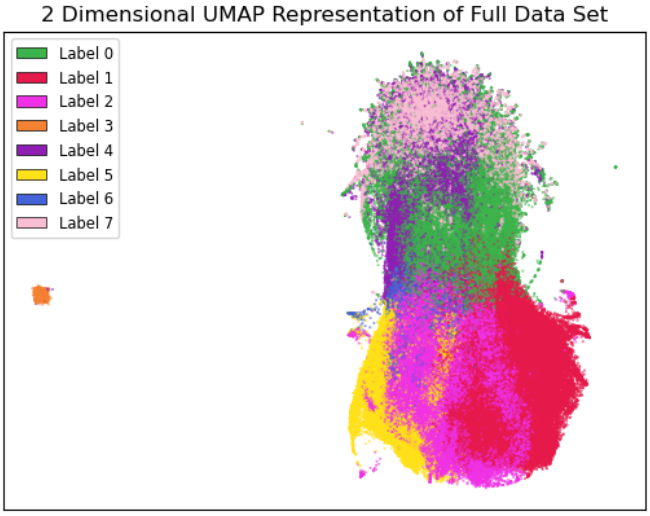


Fig. 5: *Results of k-means clustering shown over the 2 dimensional UMAP representation of the full data set.*

and .1 for the mixed split in both sets. The difference in certainty thresholds was due to any mixed term data set having an extremely large number of false positives on .01, but more reasonable results on .1.

These results show that classification of terms surrounding the Scott County HIV outbreak is achievable, but the model will need to be refined on more data. It can be seen that the mixed term split of data led to a high rate of true positives, however it quickly became much more specific to terms outside of our target class on higher epochs, with false positives dropping to lower rates. Additionally, accuracy on data_201 begins to increase between 150 and 200 epoch models for the unmixed split, so even higher epoch models might improve results further for the unmixed split. Outliers, such as in the true positives in data_121 with 100 epochs without k-means labels, can be explained by the certainty threshold. If the certainty threshold was .05 for that model, there would have been 86 true positives, and 1,129 false positives. A precise certainty threshold can be found as we test this model on other HIV related data sets and control data sets. With

	With K-Means Label							Without K-Means Label						
Epochs	Accuracy	Precision	Recall	TP	FP	TN	FN	Accuracy	Precision	Recall	TP	FP	TN	FN
50	0.9589	0.0513	0.0041	8	148	48897	1947	0.9571	0.1538	0.0266	52	286	48759	1903
100	0.9589	0.0824	0.0072	14	156	48889	1941	0.9608	0.0893	0.0026	5	51	48994	1950
150	0.6915	0.0535	0.4220	825	14602	34443	1130	0.7187	0.0451	0.3141	614	13006	36039	1341
200	0.7397	0.0388	0.2435	476	11797	37248	1479	0.7566	0.0399	0.2317	453	10912	38133	1502
50Mix	0.9881	0.9107	0.7967	1724	169	48667	440	0.9811	0.9417	0.5901	1277	79	48757	887
100Mix	0.9814	0.9418	0.5980	1294	80	48756	870	0.9823	0.9090	0.6465	1399	140	48696	765
150Mix	0.9798	0.9595	0.5471	1184	50	48786	980	0.9752	0.9934	0.4191	907	6	48830	1257
200Mix	0.9736	0.9846	0.3835	830	13	48823	1334	0.9770	0.9834	0.4658	1008	17	48819	1156

TABLE 3: *Results of the neural network run on the data_201 set. The epochs column shows the number of training epochs on the models, as well as if the words were mixed between the training and testing data, denoted by "Mix".*

	With K-Means Label							Without K-Means Label						
Epochs	Accuracy	Precision	Recall	TP	FP	TN	FN	Accuracy	Precision	Recall	TP	FP	TN	FN
50	0.9049	0.0461	0.0752	147	3041	46004	1808	0.9350	0.0652	0.0522	102	1463	47582	1853
100	0.9555	0.1133	0.0235	46	360	48685	1909	0.8251	0.0834	0.3565	697	7663	41382	1258
150	0.9554	0.0897	0.0179	35	355	48690	1920	0.9572	0.0957	0.0138	27	255	48790	1928
200	0.9496	0.0335	0.0113	22	635	48410	1933	0.9525	0.0906	0.0266	52	522	48523	1903
50Mix	0.9285	0.2973	0.5018	1086	2567	46269	1078	0.9487	0.4062	0.4501	974	1424	47412	1190
100Mix	0.9475	0.3949	0.4464	966	1480	47356	1198	0.9492	0.4192	0.5134	1111	1539	47297	1053
150Mix	0.9344	0.3112	0.4496	973	2154	46682	1191	0.9514	0.4291	0.4390	950	1264	47572	1214
200Mix	0.9449	0.3779	0.4635	1003	1651	47185	1161	0.9500	0.4156	0.4395	951	1337	47499	1213

TABLE 4: *Results of the neural network on the data_121 set. The epochs column shows the number of training epochs on the models, as well as if the words were mixed between the training and testing data, denoted by "Mix".*

enough experimentation and data, a set can be run through our pipeline and a certainty of there being a potential HIV outbreak in the region the data originated from can be generated by a future model.

Conclusion

Our results prove promising, with high accuracy and decent recall on classification of HIV/AIDS related terms, as well as potentially discovering new terms related to the outbreak. Given more HIV related data sets and control data sets, we could begin examining and generating thresholds of what might be indicative of an outbreak. To improve results, metrics for our word2vec baseline model and statistical analysis could be further explored, as well as exploring previously mentioned noise and biases from our data. Additionally, sparsity of data in earlier temporal buckets may lead to some loss of accuracy. Fine tuning hyperparameters of the alignment model through grid searching would likely even further improve these results. We predict that given more data sets containing tweets from areas and times that had similar HIV/AIDS outbreaks to Scott County, as well as control data sets that are not directly related to an HIV outbreak, we could determine a threshold of words that would define a county as potentially undergoing an HIV outbreak. With a refined pipeline and model such as this, we hope to be able to begin biosurveillance to try to prevent future outbreaks.

Future Work

Case studies of previous datasets related to other diseases and collection of more modern tweets could not only provide critical insight into relevant medical activity, but also further strengthen and expand our model and its credibility. There is a large source of data potentially related to HIV/AIDS on Twitter, so finding and collecting this data would be a crucial first step. One potent example of data could be from the 220 United States counties determined by the CDC to be considered vulnerable to HIV and/or viral hepatitis outbreaks due to injection drug use, similar to the outbreak that occurred in Scott County [VHRH+16]. Our next data set that is being studied is tweets from Cabell County, West Virginia, from January of 2018 through 2020. During this time an HIV outbreak similar to the one that took place in Scott County in 2014 occurred [AMK20]. The end goal is to create a pipeline that can perform live semantic shift analysis at set intervals of time within these counties, and classify these shifts as they happen. A future model can predict whether or not the number of terms classified as HIV related is indicative of an outbreak. If enough terms classified by our model as potentially indicative of an outbreak become detected, or if this future model predicts a possible outbreak, public health officials can be notified and the severity of a possible outbreak can be mitigated if properly handled.

Expansion into other social media platforms would increase the variety of data our model has access to, and therefore what our model is able to respond to. With the foundational model established, we will be able to focus on converting the data and addressing the differences between social networks (e.g. audience and online etiquette). Reddit and Instagram are two points of interest due to their increasing prevalence, as well as vastness of available data.

An idea for future implementation following the generation

of a generalized model would be creating a web application. The ideal audience would be medical officials and organizations, but even public or research use for trend prediction could be potent. The application would give users the ability to pick from a given glossary of medical terms, defining their own set of significant words to run our model on. Our model would then expose any potential trends or insight for the given terms in contemporary data, allowing for quicker responses to activity. Customization of the data pool could also be a feature, where tweets and other social media posts are filtered to specified geographic regions or time windows, yielding more specific results.

Additionally, we would like to reassess our embedding model to try and improve embeddings generated and our understanding of the semantic shifts. This project has been ongoing for several years, and new models, such as the use of bidirectional encoders, as in BERT [DCLT18], have proven to have high performance. BERT based models have also been used for temporal embedding studies, such as in [LMD+19], a study focused on clinical corpora. We predict that updating our pipeline to match more modern methodology can lead to more effective disease detection.

REFERENCES

[Aff05] Veteran Affairs. Glossary of HIV/AIDS terms: Veterans affairs, Dec 2005. URL: https://www.hiv.va.gov/provider/glossary/index.asp.

[AMK20] A Atkins, RP McClung, and M Kilkenny. Notes from the field: Outbreak of Human Immunodeficiency Virus infection among persons who inject drugs — Cabell County, West Virginia, 2018–2019. *Morbidity and Mortality Weekly Report*, 69(16):499–500, 2020. doi:10.15585/mmwr.mm6916a2.

[BTB14] Elia Bruni, Nam Khanh Tran, and Marco Baroni. Multimodal distributional semantics. *J. Artif. Int. Res.*, 49(1):1–47, 2014. doi:10.1613/jair.4135.

[BTKSDZ19] Arnout Boot, Erik Tjon Kim Sang, Katinka Dijkstra, and Rolf Zwaan. How character limit affects language usage in tweets. *Palgrave Communications*, 5(76), 2019. doi:10.1057/s41599-019-0280-3.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2018. doi:10.18653/v1/N19-1423.

[GC18] Gregg S Gonsalves and Forrest W Crawford. Dynamics of the HIV outbreak and response in Scott County, IN, USA, 2011–15: A modelling study. *The Lancet HIV*, 5(10), 2018. URL: https://pubmed.ncbi.nlm.nih.gov/30220531/.

[Gol17] Nicholas J. Golding. The needle and the damage done: Indiana's response to the 2015 HIV epidemic and the need to change state and federal policies regarding needle exchanges and intravenous drug users. *Indiana Health Law Review*, 14(2):173, 2017. doi:10.18060/3911.0038.

[HLJ16] William L. Hamilton, Jure Leskovec, and Dan Jurafsky. Diachronic word embeddings reveal statistical laws of semantic change. *CoRR*, abs/1605.09096, 2016. arXiv:1605.09096, doi:10.48550/arXiv.1605.09096.

[HRK15] Felix Hill, Roi Reichart, and Anna Korhonen. SimLex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, 2015. doi:10.1162/COLI_a_00237.

[LMD+19] Chen Lin, Timothy Miller, Dmitriy Dligach, Steven Bethard, and Savova Guergana. A BERT-based universal model for both within- and cross-sentence clinical temporal relation extraction. In *Proceedings of the 2nd Clinical Natural Language Processing Workshop*, pages 65–71. Association for Computational Linguistics, 2019. doi:10.18653/v1/W19-1908.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. doi:10.48550/ARXIV.1301.3781.

[PPH+16] Philip J. Peters, Pamela Pontones, Karen W. Hoover, Monita R. Patel, Romeo R. Galang, Jessica Shields, Sara J. Blosser, Michael W. Spiller, Brittany Combs, William M. Switzer, and et al. HIV infection linked to injection use of Oxymorphone in Indiana, 2014–2015. *New England Journal of Medicine*, 375(3):229–239, 2016. doi:10.1056/NEJMoa1515195.

[VHRH+16] Michelle M. Van Handel, Charles E. Rose, Elaine J. Hallisey, Jessica L. Kolling, Jon E. Zibbell, Brian Lewis, Michele K. Bohm, Christopher M. Jones, Barry E. Flanagan, Azfar-E-Alam Siddiqi, and et al. County-level vulnerability assessment for rapid dissemination of HIV or HCV infections among persons who inject drugs, United States. *JAIDS Journal of Acquired Immune Deficiency Syndromes*, 73(3):323–331, 2016. doi:10.1097/qai.0000000000001098.

[WWC+19] Bin Wang, Angela Wang, Fenxiao Chen, Yuncheng Wang, and C.-C. Jay Kuo. Evaluating word embedding models: Methods and experimental results. *APSIPA Transactions on Signal and Information Processing*, 8(1), 2019. doi:10.1017/atsip.2019.12.

[YSD+18] Zijun Yao, Yifan Sun, Weicong Ding, Nikhil Rao, and Hui Xiong. Dynamic word embeddings for evolving semantic discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining:*, WSDM '18, page 673–681, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3159652.3159703.

Design of a Scientific Data Analysis Support Platform

Nathan Martindale‡*, Jason Hite‡, Scott Stewart‡, Mark Adams‡

Abstract—Software data analytic workflows are a critical aspect of modern scientific research and play a crucial role in testing scientific hypotheses. A typical scientific data analysis life cycle in a research project must include several steps that may not be fundamental to testing the hypothesis, but are essential for reproducibility. This includes tasks that have analogs to software engineering practices such as versioning code, sharing code among research team members, maintaining a structured codebase, and tracking associated resources such as software environments. Tasks unique to scientific research include designing, implementing, and modifying code that tests a hypothesis. This work refers to this code as *an experiment*, which is defined as a software analog to physical experiments.

A software experiment manager should support tracking and reproducing individual experiment runs, organizing and presenting results, and storing and reloading intermediate data on long-running computations. A software experiment manager with these features would reduce the time a researcher spends on tedious busywork and would enable more effective collaboration. This work discusses the necessary design features in more depth, some of the existing software packages that support this workflow, and a custom developed open-source solution to address these needs.

Index Terms—reproducible research, experiment life cycle, data analysis support

Introduction

Modern science increasingly uses software as a tool for conducting research and scientific data analyses. The growing number of libraries and frameworks facilitating this work has greatly lowered the barrier to usage, allowing more researchers to benefit from this paradigm. However, as a result of the dependence on software, there is a need for more thorough integration of sound software engineering practices with the scientific process. The fragility of complex environments containing heavily interconnected packages coupled with a lack of provenance of the artifacts generated throughout the development of an experiment increases the potential for long-term problems, undetected bugs, and failure to reproduce previous analyses.

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

Fundamentally, science revolves around the ability for others to repeat and reproduce prior published works, and this has become a difficult task with many computation-based studies. Often, scientists outside of a computer science field may not have training in software engineering best practices, or they may simply disregard them because the focus of a researcher is on scientific publications rather than the analysis software itself. Lack of documentation and provenance of research artifacts and frequent failure to publish repositories for data and source code has led to a crisis in reproducibility in artificial intelligence (AI) and other fields that rely heavily on computation [SBB13], [DMR+09], [Hut18]. One study showed that quantifiably few machine learning (ML) papers document specifics in how they ran their experiments [GGA18]. This gap between established practices from the software engineering field and how computational research is conducted has been studied for some time, and the problems that can stem from it are discussed at length in [Sto18].

To mitigate these issues, computation-based research requires better infrastructure and tooling [Pen11] as well as applying relevant software engineering principles [Sto18], [Dub05] to allow data scientists to ensure their work is effective, correct, and reproducible. In this paper we focus on the ability to manage reproducible workflows for scientific experiments and data analyses. We discuss the features that software to support this might require, compare some of the existing tools that address them, and finally present the open-source tool Curifactory which incorporates the proposed design elements.

Related Work

Reproducibility of AI experiments has been separated into three different degrees [GK18]: *Experiment reproduciblity*, or repeatability, refers to using the same code implementation with the same data to obtain the same results. *Data reproducibility*, or replicability, is when a different implementation with the same data outputs the same results. Finally, *method reproducibility* describes when a different implementation with different data is able to achieve consistent results. These degrees are discussed in [GGA18], comparing the implications and trade-offs on the amount of work for the original researcher versus an external researcher, and the degree of generality afforded by a reproduced implementation. A repeatable experiment places the greatest burden on the original researcher, requiring the full codebase and experiment to be sufficiently documented and published so that a peer is able to correctly repeat it. At the other end of the spectrum, method reproducibility demands the greatest burden on the external researcher, as they must implement and run the experiment from scratch. For the remainder of this paper, we refer

to "reproducibility" as experiment reproducibility (repeatability). Tooling that is able to assist with documentation and organization of a published experiment reduces the amount of work for the original researcher and still allows for the lowest level of burden to external researchers to verify and extend previous work.

In an effort to encourage better reproducibility based on datasets, the Findable, Accessible, Interoperable, and Reusable (FAIR) data principles [WDA+16] were established. These principles recommend that data should have unique and persistent identifiers, use common standards, and provide rich metadata description and provenance, allowing both humans and machines to effectively parse them. These principles have been extended more broadly to software [LGK+20], computational workflows [GCS+20], and to entire data pipelines [MLC+21].

Various works have surveyed software engineering practices and identified practices that provide value in scientific computing contexts, including various forms of unit and regression testing, proper source control usage, formal verification, bug tracking, and agile development methods [Sto18], [Dub05]. In particular, [Sto18] described many concepts from agile development as being well suited to an experimental context, where the current knowledge and goals may be fairly dynamic throughout the project. They noted that although many of these techniques could be directly applied, some required adaptation to make sense in the scientific software domain.

Similar to this paper, two other works [DGST09], [WWG21] discuss sets of design aspects and features that a workflow manager would need. Deelman et al. describe the life cycle of a workflow as composition, mapping, execution, and provenance capture [DGST09]. A workflow manager must then support each of these aspects. Composition is how the workflow is constructed, such as through a graphical interface or with a text configuration file. Mapping and execution are determining the resources to be used for a workflow and then utilizing those resources to run it, including distributing to cloud compute and external representational state transfer (REST) services. This also refers to scheduling subworkflows/tasks to reuse intermediate artifacts as available. Provenance, which is crucial for enabling repeatability, is how all artifacts, library versions, and other relevant metadata are tracked during the execution of a workflow.

Wratten, Wilm, and Göke surveyed many bioinformatics pipline and workflow management tools, listing the challenges that tooling should address: data provenance, portability, scalability, and re-entrancy [WWG21]. Provenance is defined the same way as in [DGST09], and further states the need for generating reports that include the tracking information and metadata for the associated experiment run. Portability—allowing set up and execution of an experiment in a different environment—can be a challenge because of the dependency requirements of a given system and the ease with which the environment can be specified and reinitialized on a different machine or operating system. Scalability is important especially when large scale data, many compute-heavy steps, or both are involved throughout the workflow. Scalability in a manager involves allowing execution on a high-performance computing (HPC) system or with some form of parallel compute. Finally they mention re-entrancy, or the ability to resume execution of a compute step from where it last stopped, preventing unnecessary recomputation of prior steps.

One area of the literature that needs further discussion is the design of automated provenance tracking systems. Existing workflow management tools generally require source code modifications to take full advantage of all features. This can entail a significant learning curve and places additional burden on the researcher. To address this, some sources propose automatic documentation of experiments and code through static source code analysis [NFP+20], [Red19].

Beyond the preexisting body of knowledge about software engineering principles, other works [SNTH13], [KHS09] describe recommended rules and practices to follow when conducting computation-based research. These include avoiding manual data manipulation in favor of scripted changes, keeping detailed records of how results are produced (manual provenance), tracking the versions of libraries and programs used, and tracking random seeds. Many of these ideas can be assisted or encapsulated through appropriate infrastructure decisions, which is the premise on which this work bases its software reviews.

Although this paper focuses on the scientific workflow, a growing related field tackles many of the same issues from an industry standpoint: machine learning operations (MLOps) [Goy20]. MLOps, an ML-oriented version of DevOps, is concerned with supporting an entire data science life cycle, from data acquisition to deployment of a production model. Many of the same challenges are present, reproducibility and provenance are crucial in both production and research workflows [RMRO21]. Infrastructure, tools, and practices developed for MLOps may also hold value in the scientific community.

A taxonomy for ML tools that we reference throughout this work is from [QCL21], which describes a characterization of tools consisting of three primary categories: general, analysis support, and reproducibility support, each of which is further subdivided into aspects to describe a tool. For example, these subaspects include data visualization, web dashboard capabilities, experiment logging, and the interaction modes the tool supports, such as a command line interface (CLI) or application programming interface (API).

Design Features

We combine the two sets of capabilities from [DGST09] and [WWG21] with the taxonomy from [QCL21] to propose a set of six design features that are important for an experiment manager. These include orchestration, parameterization, caching, reproducibility, reporting, and scalability. The crossover between these proposed feature sets are shown in Table 1. We expand on each of these in more depth in the subsections below.

Orchestration

Orchestration of an experiment refers to the mechanisms used to chain and compose a sequence of smaller logical steps into an overarching pipeline. This provides a higher-level view of an experiment and helps abstract away some of the implementation details. Operation of most workflow managers is based on a directed acyclic graph (DAG), which specifies the stages/steps as nodes and the edges connecting them as their respective inputs and outputs. The intent with orchestration is to encourage designing distinct, reusable steps that can easily be composed in different ways to support testing different hypotheses or overarching experiment runs. This allows greater focus on the design of the experiments than the implementation of the underlying functions that the experiments consist of. As discussed in the taxonomy [QCL21], pipeline creation can consist of a combination of scripts, configuration files, or a visual tool. This aspect falls within the composition capability discussed in [DGST09].

This work	[DGST09]	[WWG21]	Taxonomy [QCL21]
Orchestration	Composition	—	Reproducibility/pipeline creation
Parameterization	—	—	—
Caching	—	Re-entrancy	—
Reproducibility	Provenance	Provenance, portability	Reproducibility
Reporting	—	—	Analysis/visualization, web dashboard
Scalability	Mapping, execution	Scalability	Analysis/computational resources

TABLE 1: *Comparing design features listed in various works.*

Parameterization

Parameterization specifies how a compute pipeline is customized for a particular run by passing in configuration values to change aspects of the experiment. The ability to customize analysis code is crucial to conducting a compute-based experiment, providing a mechanism to manipulate a variable under test to verify or reject a hypothesis.

Conventionally, parameterization is done either through specifying parameters in a CLI call or by passing configuration files in a format like JSON or YAML. As discussed in [DGST09], parameterization sometimes consists of more complicated needs, such as conducting parameter sweeps or grid searches. There are libraries dedicated to managing parameter searches like this, such as hyperopt [BYC13] used in [RMRO21].

Although not provided as a design capability in the other works, we claim the mechanisms provided for parameterization are important, as these mechanisms are the primary way to configure, modify, and vary experiment execution without explicitly changing the code itself or modifying hard-coded values. This means that a recorded parameter set can better "describe" an experiment run, increasing provenance and making it easier for another researcher to understand what pieces of an experiment can be readily changed and explored.

Some support is provided for this in [DGST09], stating that the necessity of running many slight variations on workflows sometimes leads to the creation of ad hoc scripts to generate the variants, which leads to increased complexity in the organization of the codebase. Improved mechanisms to parameterize the same workflow for many variants helps to manage this complexity.

Caching

Refining experiment code and finding bugs is often a lengthy iterative process, and removing the friction of constantly rerunning all intermediate steps every time an experiment is wrong can improve efficiency. Caching values between each step of an experiment allows execution to resume at a certain spot in the pipeline, rather than starting from scratch every time. This is defined as *re-entrancy* in [WWG21].

In addition to increasing the speed of rerunning experiments and running new experiments that combine old results for analysis, caching is useful to help find and debug mistakes throughout an experiment. Cached outputs from each step allow manual interrogation outside of the experiment. For example, if a cleaning step was implemented incorrectly and a user noticed an invalid value in an output data table, they could use a notebook to load and manipulate the intermediate artifact tables for that data to determine what stage introduced the error and what code should be used to correctly fix it.

Reproducibility

Mechanisms for reproducibility are one of the most important features for a successful data analysis support platform. Reproducibility is challenging because of the complexity of constantly evolving codebases, complicated and changing dependency graphs, and inconsistent hardware and environments. Reproducibility entails two subcomponents: provenance and portability. This falls under the provenance aspect from [DGST09], both data provenance and portability from [WWG21], and the entire reproducibility support section of the taxonomy [QCL21].

Data provenance is about tracking the history, configuration, and steps taken to produce an intermediate or final data artifact. In ML this would include the cleaning/munging steps used and the intermediate tables created in the process, but provenance can apply more broadly to any type of artifact an experiment may produce, such as ML models themselves, or "model provenance" [SH18]. Applying provenance beyond just data is critical, as models may be sensitive to the specific sets of training data and conditions used to produce them [Hut18]. This means that everything required to directly and exactly reproduce a given artifact is recorded, such as the manipulations applied to its predecessors and all hyperparameters used within those manipulations.

Portability refers to the ability to take an experiment and execute it outside of the initial computing environment it was created in [WWG21]. This can be a challenge if all software dependency versions are not strictly defined, or when some dependencies may not be available in all environments. Minimally, allowing portability requires keeping explicit track of all packages and the versions used. A 2017 study [OBA17] found that even this minimal step is rarely taken. Another mechanism to support portability is the use of containerization, such as with Docker or Podman [SH18].

Reporting

Reporting is an important step for analyzing the results of an experiment, through visualizations, summaries, comparisons of results, or combinations thereof. As a design capability, *reporting* refers to the mechanisms available for the system to export or retrieve these results for human analysis. Although data visualization and analysis can be done manually by the scientist, tools to assist with making these steps easier and to keep results organized are valuable from a project management standpoint. Mechanisms for this might include a web interface for exploring individual or multiple runs. Under the taxonomy [QCL21], this falls primarily within analysis support, such as data visualization or a web dashboard.

Scalability

Many data analytic problems require large amounts of space and compute resources, often beyond what can be handled on an individual machine. To efficiently support running a large experiment, mechanisms for scaling execution are important and could include anything from supporting parallel computation on an experiment or stage level, to allowing the execution of jobs on remote machines or within an HPC context. This falls within both mapping and execution from [DGST09], the scalability aspect from [WWG21], and the computational resources category within the taxonomy [QCL21].

Existing Tools

A wide range of pipeline and workflow tools have been developed to support many of these design features, and some of the more common examples include DVC [KPP+22] and MLFlow [MLf22]. We briefly survey and analyze a small sample of these tools to demonstrate the diversity of ideas and their applicability in different situations. Table 2 compares the support of each design feature by each tool.

DVC

DVC [KPP+22] is a Git-like version control tool for datasets. Orchestration is done by specifying *stages*, or runnable script commands, either in YAML or directly on the CLI. A stage is specified with output file paths and input file paths as dependencies, allowing an implicit pipeline or DAG to form, representing all the processing steps. Parameterization is done by defining within a YAML file what the possible parameters are, along with the default values. When running the DAG, parameters can be customized on the CLI. Since inputs and outputs are file paths, caching and re-entrancy come for free, and DVC will intelligently determine if certain stages do not need to be re-computed.

A saved experiment or state is frozen into each commit, so all parameters and artifacts are available at any point. No explicit tracking of the environment (e.g., software versions and hardware info) is present, but this could be manually included by tracking it in a separate file. Reporting can be done by specifying per-stage metrics to track in the YAML configuration. The CLI includes a way to generate HTML files on the fly to render requested plots. There is also an external "Iterative Studio" project, which provides a live web dashboard to view continually updating HTML reports from DVC. For scalability, parallel runs can be achieved by queuing an experiment multiple times in the CLI.

MLFlow

MLFlow [MLf22] is a framework for managing the entire life cycle of an ML project, with an emphasis on scalability and deployment. It has no specific mechanisms for orchestration, instead allowing the user to intersperse MLFlow API calls in an existing codebase. Runnable scripts can be provided as entry points into a configuration YAML, along with the parameters that can be provided to it. Parameters are changed through the CLI. Although MLFlow has extensive capabilities for tracking artifacts, there are no automatic re-entrancy methods. Reproducibility is a strong feature, and provenance and portability are well supported. The tracking module provides provenance by recording metadata such as the Git commit, parameters, metrics, and any user-specified artifacts in the code. Portability is done by allowing the environment for an entry point to be specified as a Conda environment or Docker container. MLFlow then ensures that the environment is set up and active before running. The CLI even allows directly specifying a GitHub link to an mlflow-enabled project to download, set up, and then run the associated experiment. For reporting, the MLFlow tracking UI lets the user view and compare various runs and their associated artifacts through a web dashboard. For scalability, both distributed storage for saving/loading artifacts as well as execution of runs on distributed clusters is supported.

Sacred

Sacred [GKC+17] is a Python library and CLI tool to help organize and reproduce experiments. Orchestration is managed through the use of Python decorators, a "main" for experiment entry point functions and "capture" for parameterizable functions, where function arguments are automatically populated from the active configuration when called. Parameterization is done directly in Python through applying a config decorator to a function that assigns variables. Configurations can also be written to or read from JSON and YAML files, so parameters must be simple types. Different observers can be specified to automatically track much of the metadata, environment information, and current parameters, and within the code the user can specify additional artifacts and resources to track during the run. Each run will store the requested outputs, although there is no re-entrant use of these cached values. Portability is supported through the ability to print the versions of libraries needed to run a particular experiment. Reporting can be done through a specific type of observer, and the user can provide custom templated reports that are generated at the end of each run.

Kedro

Kedro [ABC+22] is another Python library/CLI tool for managing reproducible and modular experiments. Orchestration is particularly well done with "node" and "pipeline" abstractions, a node referring to a single compute step with defined inputs and outputs, and a pipeline implemented as an ordered list of nodes. Pipelines can be composed and joined to create an overarching workflow. Possible parameters are defined in a YAML file and either set in other parameter files or configured on the CLI. Similar to MLFlow, while tracking outputs are cached, there's no automatic mechanism for re-entrancy. Provenance is achieved by storing user-specified metrics and tracked datasets for each run, and it has a few different mechanisms for portability. This includes the ability to export an entire project into a Docker container. A separate Kedro-Viz tool provides a web dashboard to show a map of experiments, as well as showing each tracked experiment run and allowing comparison of metrics and outputs between them. Projects can be deployed into several different cloud providers, such as Databricks and Dask clusters, allowing for several options for scalability.

Curifactory

Curifactory [MHSA22] is a Python API and CLI tool for organizing, tracking, reproducing, and exporting computational research experiments and data analysis workflows. It is intended primarily for smaller teams conducting research, rather than production-level or large-scale ML projects. Curifactory is available on GitHub[1] with an open-source BSD-3-Clause license. Below, we describe the mechanisms within Curifactory to support each of the six capabilities, and compare it with the tools discussed above.

1. https://github.com/ORNL/curifactory

	Orchestration	Parameterization	Caching	Provenance	Portability	Reporting	Scalability
DVC	+	+	++	+	+	+	+
MLFlow		+	*	++	++	++	++
Sacred	+	++	*	++	+	+	
Kedro	+	+	*	+	++	++	++
Curifactory	+	++	++	++	++	+	+

TABLE 2: *Supported design features in each tool. Note, + indicates that a feature is supported, ++ indicates very strong support, and * indicates tooling that supports caching artifacts as a provenance tool but does not provide a mechanism for automatically reloading cached values as a form of re-entrancy.*

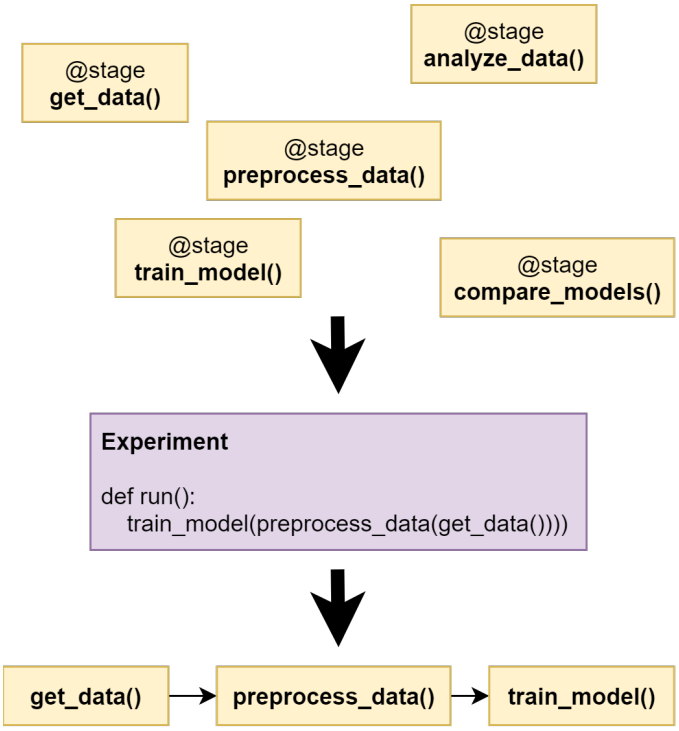


Fig. 1: *Stages are composed into an experiment.*

Orchestration

Curifactory provides several abstractions, the lowest level of which is a *stage*. A stage is a function that takes a defined set of input variable names, a defined set of output variable names, and an optional set of caching strategies for the outputs. Stages are similar to Kedro's nodes but implemented with `@stage()` decorators on the target function rather than passing the target function to a `node()` call. One level up from a stage is an *experiment*: an experiment describes the orchestration of these stages as shown in Figure 1, functionally chaining them together without needing to explicitly manage what variables are passed between the stages.

```
@stage(inputs=None, outputs=["data"])
def load_data(record):
    # every stage has the currently active record
    # passed to it, which contains the "state", or
    # all previous output values associated with
    # the current argset, as defined in the
    # Parameterization section
    # ...

@stage(inputs=["data"], outputs=["model", "stats"])
def train_model(record, data):
    # ...

@stage(inputs=["model"], outputs=["results"])
def test_model(record, model):
    # ...

def run(argsets, manager):
    """An example experiment definition.

    The primary intent of an experiment is to run
    each set of arguments through the desired
    stages, in order to compare results at the end.
    """
    for argset in argsets:
        # A record is the "pipeline state"
        # associated with each set of arguments.
        # Stages take and return a record,
        # automatically handling pushing and
        # pulling inputs and outputs from the
        # record state.
        record = Record(manager, argsets)
        test_model(train_model(load_data(record)))
```

Parameterization

Parameterization in Curifactory is done directly in Python scripts. The user defines a dataclass with the parameters they need throughout their various stages in order to customize the experiment, and they can then define parameter files that each return one or more instances of this arguments class. All stages in an experiment are automatically given access to the current argument set in use while an experiment is running.

While configuration can also be done directly in Python in Sacred, Curifactory makes a different trade-off: A parameter file or `get_params()` function in Curifactory returns an array of one or more argument sets, and arguments can directly include complex Python objects. Unlike Sacred, this means Curifactory cannot directly translate back and forth from static configuration files, but in exchange allows for grid searches to be defined directly and easily in a single parameter file, as well as allowing argument sets to be composed or even inherit from other argument set instances. Importantly, Curifactory can still encode representations of arguments into JSON for provenance, but this is a one directional transformation.

This approach allows a great deal of flexibility, and is valuable in experiments where a large range of parameters need to be tested or there is significant repetition among parameter sets. For example, in an experiment testing different effects of model training hyperparameters, there may be several parameter files meant to vary only the arguments needed for model training while using the same base set of data cleaning arguments. Composing these parameter sets from a common imported set means that any subsequent changes to the data cleaning arguments only need to

be modified in one place, rather than each individual parameter file.

```
@dataclass
class MyArgs(curifactory.ExperimentArgs):
    """Define the possible arguments needed in the
    stages."""
    random_seed: int = 42
    train_test_ratio: float = 0.8
    layers: tuple = (100,)
    activation: str = "relu"

def get_params():
    """Define a simple grid search: return
    many arguments instances for testing."""
    args = []
    layer_sizes = [10, 20, 50, 100]
    for size in layer_sizes:
        args.append(MyArgs(name=f"network_{size}",
            layers=(size,)))
    return args
```

Caching

Curifactory supports per-stage caching, similar to memoization, through a set of easy-to-use caching strategies. When a stage executes, it uses the specified cache mechanism to store the stage outputs to disk, with a filename based on the experiment, stage, and a hash of the arguments. When the experiment is re-executed, if it finds an existing output on disk based on this name, it short-circuits the stage computation and simply reloads the previously cached files, allowing a form of re-entrancy. Adding this caching ability to a stage is done through simply providing the list of caching strategies to the stage decorator, one for each output:

```
@stage(
    inputs=["data"],
    outputs=["training_set", "testing_set"],
    cachers=[PandasCSVCacher]*2
):
def split_data(record, data):
    # stage definition
```

Reproducibility

As mentioned before, reproducibility consists of tracking provenance and metadata of artifacts as well as providing a means to set up and repeat an experiment in a different compute environment. To handle provenance, Curifactory automatically records metadata for every experiment run executed, including a logfile of the console output, current Git commit hash, argument sets used and the rendered versions of those arguments, and the CLI command used to start the run. The final reports from each run also include a graphical representation of the stage DAG, and shows each output artifact and what its cache file location is.

Curifactory has two mechanisms to fully track and export an experiment run. The first is to execute a "full store" run, which creates a single exported folder containing all metadata mentioned above, along with a copy of every cache file created, the output run report (mentioned below), as well as a Python requirements.txt and Conda environment dump, containing a list of all packages in the environment and their respective versions. This run folder can then be distributed. Reproducing from the folder consists of setting up an environment based on the Conda/Python dependencies as needed, and running the experiment command using the exported folder as the cache directory.

The second mechanism is a command to create a Docker container that includes the environment, entire codebase, and artifact cache for a specific experiment run. Curifactory comes with a default Dockerfile for this purpose, and running the experiment with the Docker flag creates an image that exposes a Jupyter notebook to repeat the run and keep the artifacts in memory, as well as a file server pointing to the appropriate cache for manual exploration and inspection. Directly reproducing the experiment can be done either through the exposed notebook or by running the Curifactory experiment command inside of the image.

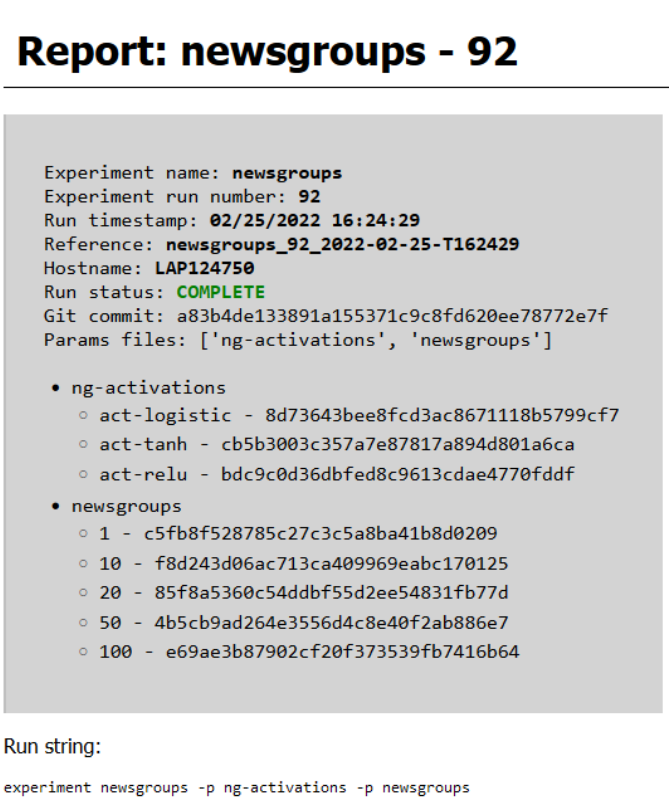


Fig. 2: *Metadata block at the top of a report.*

Reporting

While Curifactory does not run a live web dashboard like MLFlow, DVC's Iterative Studio, and Kedro-viz, every experiment run outputs an HTML experiment report and updates a top-level index HTML page linking to the new report, which can be browsed from a file manager or statically served if running from an external compute resource. Although simplistic, this reduces the dependencies and infrastructure needed to achieve a basic level of reporting, and produces stand-alone folders for consumption outside of the original environment if needed.

Every report from Curifactory includes all relevant metadata mentioned above, including the machine host name, experiment sequential run number, Git commit hash, parameters, and command line string. Stage code can add user-defined objects to output in each report, such as tables, figures, and so on. Curifactory comes with a default set of helpers for several basic types of output visualizations, including basic line plots, entire Matplotlib figures, and dataframes.

The output report also contains a graphical representation of the DAG for the experiment, rendered using Graphviz, and shows the artifacts produced by each stage and the file path where they are cached. An example of some of the components of this report are rendered in figures 2, 3, 4, and 5.

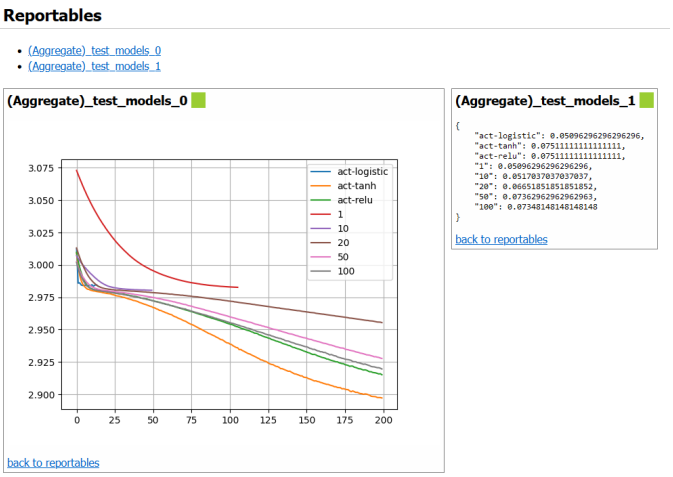


Fig. 3: *User-defined objects to report ("reportables").*

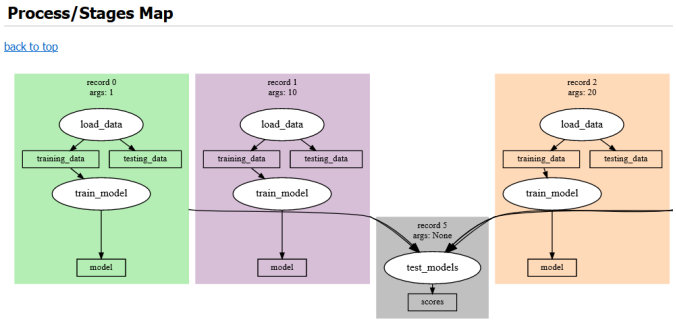


Fig. 4: *Graphviz rendering of experiment DAG. Each large colored area represents a single record associated with a specific argset. White ellipses are stages, and the blocks in between them are the input and output artifacts.*

Scalability

Curifactory has no integrated method of executing portions of jobs on external compute resources like Kedro and MLFlow, but it does allow local multi-process parallelization of parameter sets. When an experiment run would entail executing a series of stages for each argument set in series, Curifactory can divide the collection of argument sets into one subcollection per process, and runs the experiment in parallel on each subcollection. By taking advantage of the caching mechanism, when all parallel runs complete, the experiment reruns in a single process to aggregate all of the precached values into a single report.

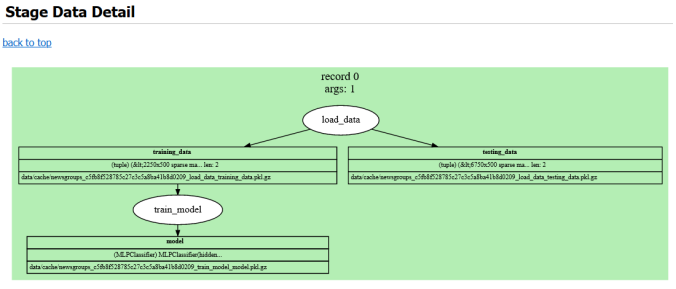


Fig. 5: *Graphviz rendering of each record in more depth, showing cache file paths and artifact data types.*

Conclusion

The complexity in modern software, environments, and data analytic approaches threaten the reproducibility and effectiveness of computation-based studies. This has been compounded by the lack of standardization in infrastructure tools and software engineering principles applied within scientific research domains. While many novel tools and systems are in development to address these shortcomings, several design critieria must be met, including the ability to easily compose and orchestrate experiments, parameterize them to manipulate variables under test, cache intermediate artifacts, record provenance of all artifacts and allow the software to port to other systems, produce output visualizations and reports for analysis, and scale execution to the resource requirements of the experiment. We developed Curifactory to address these criteria specifically for small research teams running Python based experiments.

Acknowledgements

The authors would like to acknowledge the US Department of Energy, National Nuclear Security Administration's Office of Defense Nuclear Nonproliferation Research and Development (NA-22) for supporting this work.

References

[ABC+22] Sajid Alam, Lorena Bălan, Gabriel Comym, Yetunde Dada, Ivan Danov, Lim Hoang, Rashida Kanchwala, Jiri Klein, Antony Milne, Joel Schwarzmann, Merel Theisen, and Susanna Wong. Kedro. https://kedro.org/, March 2022.

[BYC13] James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning*, pages 115–123. PMLR, February 2013.

[DGST09] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25:524–540, May 2009. doi:10.1016/j.future.2008.06.012.

[DMR+09] David L. Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. Reproducible Research in Computational Harmonic Analysis. *Computing in Science Engineering*, 11(1):8–18, January 2009. doi:10.1109/MCSE.2009.15.

[Dub05] P.F. Dubois. Maintaining correctness in scientific programs. *Computing in Science Engineering*, 7(3):80–85, May 2005. doi:10.1109/MCSE.2005.54.

[GCS+20] Carole Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R. Crusoe, Kristian Peters, and Daniel Schober. FAIR Computational Workflows. *Data Intelligence*, 2(1-2):108–121, January 2020. doi:10.1162/dint_a_00033.

[GGA18] Odd Erik Gundersen, Yolanda Gil, and David W. Aha. On Reproducible AI: Towards Reproducible Research, Open Science, and Digital Scholarship in AI Publications. *AI Magazine*, 39(3):56–68, September 2018. doi:10.1609/aimag.v39i3.2816.

[GK18] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the Art: Reproducibility in Artificial Intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018. doi:10.1609/aaai.v32i1.11503.

[GKC+17] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In *Proceedings of the 16th Python in Science Conference*, pages 49–56, Austin, Texas, 2017. SciPy. doi:10.25080/shinma-7f4c6e7-008.

[Goy20] A. Goyal. Machine learning operations, 2020.

[Hut18] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, February 2018. doi:10.1126/science.359.6377.725.

[KHS09] Diane Kelly, Daniel Hook, and Rebecca Sanders. Five Recommended Practices for Computational Scientists Who Write Software. *Computing in Science Engineering*, 11(5):48–53, September 2009. doi:10.1109/MCSE.2009.139.

[KPP+22] Ruslan Kuprieiev, Saugat Pachhai, Dmitry Petrov, Paweł Redzyński, Casper da Costa-Luis, Peter Rowlands, Alexander Schepanovski, Ivan Shcheklein, Batuhan Taskaya, Jorge Orpinel, Gao, Fábio Santos, David de la Iglesia Castro, Aman Sharma, Zhanibek, Dani Hodovic, Nikita Kodenko, Andrew Grigorev, Earl, Nabanita Dash, George Vyshnya, maykulkarni, Max Hora, Vera, Sanidhya Mangal, Wojciech Baranowski, Clemens Wolff, and Kurian Benoy. DVC: Data Version Control - Git for Data & Models. Zenodo, April 2022. doi:10.5281/zenodo.6417224.

[LGK+20] Anna-Lena Lamprecht, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, Stephanie van de Sandt, Jon Ison, Paula Andrea Martinez, Peter McQuilton, Alfonso Valencia, Jennifer Harrow, Fotis Psomopoulos, Josep Ll Gelpi, Neil Chue Hong, Carole Goble, and Salvador Capella-Gutierrez. Towards FAIR principles for research software. *Data Science*, 3(1):37–59, January 2020. doi:10.3233/DS-190026.

[MHSA22] Nathan Martindale, Jason Hite, Scott L. Stewart, and Mark Adams. Curifactory. https://github.com/ORNL/curifactory, March 2022.

[MLC+21] Sonia Natalie Mitchell, Andrew Lahiff, Nathan Cummings, Jonathan Hollocombe, Bram Boskamp, Dennis Reddyhoff, Ryan Field, Kristian Zarebski, Antony Wilson, Martin Burke, Blair Archibald, Paul Bessell, Richard Blackwell, Lisa A. Boden, Alys Brett, Sam Brett, Ruth Dundas, Jessica Enright, Alejandra N. Gonzalez-Beltran, Claire Harris, Ian Hinder, Christopher David Hughes, Martin Knight, Vino Mano, Ciaran McMonagle, Dominic Mellor, Sibylle Mohr, Glenn Marion, Louise Matthews, Iain J. McKendrick, Christopher Mark Pooley, Thibaud Porphyre, Aaron Reeves, Edward Townsend, Robert Turner, Jeremy Walton, and Richard Reeve. FAIR Data Pipeline: Provenance-driven data management for traceable scientific workflows. *arXiv:2110.07117 [cs, q-bio]*, October 2021. arXiv:2110.07117.

[MLf22] MLflow: A Machine Learning Lifecycle Platform. https://mlflow.org/, April 2022.

[NFP+20] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 1542–1551, New York, NY, USA, August 2020. Association for Computing Machinery. doi:10.1145/3394486.3403205.

[OBA17] Babatunde K. Olorisade, Pearl Brereton, and Peter Andras. Reproducibility in Machine Learning-Based Studies: An Example of Text Mining. In *Reproducibility in ML Workshop, 34th International Conference on Machine Learning*, ICML 2017, June 2017.

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

[QCL21] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. A Taxonomy of Tools for Reproducible Machine Learning Experiments. In *AIxIA 2021 Discussion Papers, 20th International Conference of the Italian Association for Artificial Intelligence*, pages 65–76, 2021.

[Red19] Sergey Redyuk. Automated Documentation of End-to-End Experiments in Data Science. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2076–2080, April 2019. doi:10.1109/ICDE.2019.00243.

[RMRO21] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences*, 11(19):8861, January 2021. doi:10.3390/app11198861.

[SBB13] Victoria Stodden, Jonathan Borwein, and David H. Bailey. Publishing Standards for Computational Science: "Setting the Default to Reproducible". Pennsylvania State University, 2013.

[SH18] Peter Sugimura and Florian Hartl. Building a Reproducible Machine Learning Pipeline. *arXiv:1810.04570 [cs, stat]*, October 2018. arXiv:1810.04570.

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

[Sto18] Tim Storer. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Computing Surveys*, 50(4):1–32, July 2018. doi:10.1145/3084225.

[WDA+16] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C. 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016. doi:10.1038/sdata.2016.18.

[WWG21] Laura Wratten, Andreas Wilm, and Jonathan Göke. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature Methods*, 18(10):1161–1168, October 2021. doi:10.1038/s41592-021-01254-9.

The Geoscience Community Analysis Toolkit: An Open Development, Community Driven Toolkit in the Scientific Python Ecosystem

Orhan Eroglu[‡*], Anissa Zacharias[‡], Michaela Sizemore[‡], Alea Kootz[‡], Heather Craker[‡], John Clyne[‡]

https://www.youtube.com/watch?v=34zFGkDwJPc

Abstract—The Geoscience Community Analysis Toolkit (GeoCAT) team develops and maintains data analysis and visualization tools on structured and unstructured grids for the geosciences community in the Scientific Python Ecosystem (SPE). In response to dealing with increasing geoscientific data sizes, GeoCAT prioritizes scalability, ensuring its implementations are scalable from personal laptops to HPC clusters. Another major goal of the GeoCAT team is to ensure community involvement throughout the whole project lifecycle, which is realized through an open development mindset by encouraging users and contributors to get involved in decision-making. With this model, we not only have our project stack open-sourced but also ensure most of the project assets that are directly related to the software development lifecycle are publicly accessible.

Index Terms—data analysis, geocat, geoscience, open development, open source, scalability, visualization

Introduction

The Geoscience Community Analysis Toolkit (GeoCAT) team, established in 2019, leads the software engineering efforts of the National Center for Atmospheric Research (NCAR) "Pivot to Python" initiative [Geo19]. Before then, NCAR Command Language (NCL) [BBHH12] was developed by NCAR as an interpreted, domain-specific language that was aimed to support the analysis and visualization needs of the global geosciences community. NCL had been serving several tens of thousands of users for decades. It is still available for use but has not been actively developed as it has been in maintenance mode.

The initiative had an initial two-year roadmap with major milestones being: (1) Replicating NCL's computational routines in Python, (2) training and support for transitioning NCL users into Python, and (3) moving tools into an open development model. GeoCAT aims to create scalable data analysis and visualization tools on structured and unstructured grids for the geosciences community in the SPE. The GeoCAT team is committed to open development, which helps the team prioritize community involvement at any level of the project lifecycle alongside having the whole software stack open-sourced.

* *Corresponding author: oero@ucar.edu*
‡ *National Center for Atmospheric Research*

GeoCAT has seven Python tools for geoscientific computation and visualization. These tools are built upon the Pangeo [HRA18] ecosystem. In particular, they rely on Xarray [HH17], and Dask [MR15], as well as they are compatible with Numpy and use Jupyter Notebooks for demonstration purposes. Dask compatibility allows the GeoCAT functions to scale from personal laptops to high performance computing (HPC) systems such as NCAR's Casper, Cheyenne, and upcoming Derecho clusters [CKZ+22]. Additionally, GeoCAT also utilizes Numba, an open source just-in-time (JIT) compiler [LPS15], to translate Python and NumPy code into machine codes in order to get faster executions wherever possible. GeoCAT's visualization components rely on Matplotlib [Hun07] for most of the plotting functionalities, Cartopy [Met15] for projections, as well as the Datashader and Holoviews stack [Anaa] for big data rendering. Figure 1 shows these technologies with their essential roles around GeoCAT.

Briefly, GeoCAT-comp houses computational operators for applications ranging from regridding and interpolation, to climatology and meteorology. GeoCAT-examples provides over 140 publication-quality plotting scripts in Python for Earth sciences. It also houses Jupyter notebooks with high-performance, interactive plots that enable features such as pan and zoom on fine-resolution, unstructured geoscience data (e.g. ~3 km data rendered within a few tens of seconds to a few minutes on personal laptops). This is achieved by making use of the connectivity information in the unstructured grid and rendering data via the Datashader and Holoviews ecosystem [Anaa]. GeoCAT-viz enables higher-level implementation of Matplotlib and Cartopy plotting capabilities through its variety of easy to use visualization convenience functions for GeoCAT-examples. GeoCAT also maintains WRF-Python (Weather Research and Forecasting), which works with WRF-ARW model output and provides diagnostic and interpolation routines.

GeoCAT was recently awarded Project Raijin, which is an NSF EarthCube-funded effort [NSF21] [CEMZ21]. Its goal is to enhance the open-source analysis and visualization tool landscape by developing community-owned, sustainable, scalable tools that facilitate operating on unstructured climate and global weather data in the SPE. Throughout this three-year project, GeoCAT will work on the development of data analysis and visualization functions that operate directly on the native grid as well as establish an active community of user-contributors.

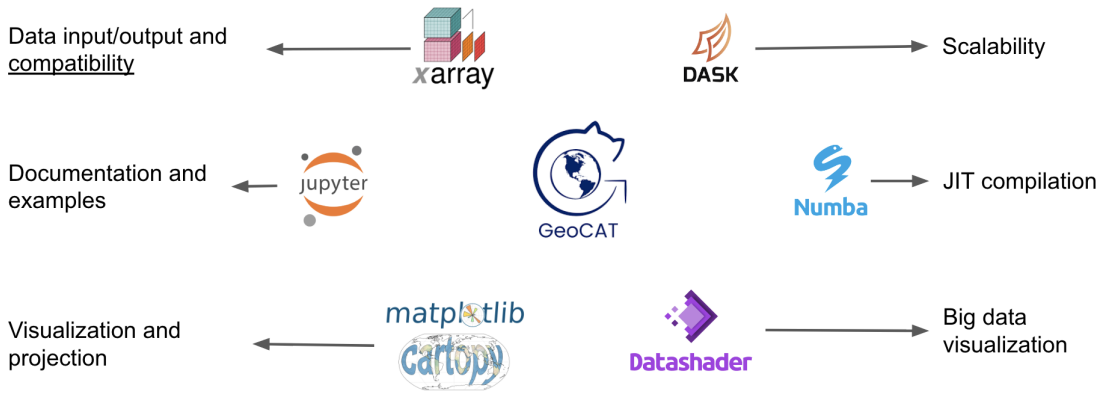


Fig. 1: *The core Python technologies on which GeoCAT relies on*

This paper will provide insights about GeoCAT's software stack and current status, team scope and near-term plans, open development methodology, as well as current pathways of community involvement.

GeoCAT Software

The GeoCAT team develops and maintains several open-source software tools. Before describing those tools, it is vital to explain in detail how the team implements the continuous integration and continuous delivery/deployment (CI/CD) in consistence for all of those tools.

Continuous Integration and Continuous Delivery/Deployment (CI/CD)

GeoCAT employs a continuous delivery model, with a monthly package release cycle on package management systems and package indexes such as Conda [Anab] and PyPI [Pyt]. This model helps the team make new functions available as soon as they are implemented and address potential errors quickly. To assist this process, the team utilizes multiple tools throughout GitHub assets to ensure automation, unit testing and code coverage, as well as licensing and reproducibility. Figure 2, for example, shows the set of badges displaying the near real-time status of each CI/CD implementation in the GitHub repository homepage from one of our software tools.

CI build tests of our repositories are implemented and automated (for pushed commits, pull requests, and daily scheduled execution) via GitHub Actions workflows [Git], with the *CI* badge shown in Figure 2 displaying the status (i.e. pass or fail) of those workflows. Similarly, the *CONDA-BUILDS* badge shows if the conda recipe works successfully for the repository. The Python package "codecov" [cod] analyzes the percentage of code coverage from unit tests in the repository. Additionally, the overall results as well as details for each code script can be seen via the *COVERAGE* badge. Each of our software repositories has a corresponding documentation page that is populated mostly-automatically through the Sphinx Python documentation generator [Bra21] and published through ReadTheDocs [rea] via an automated building and versioning schema. The *DOCS* badge provides a link to the documentation page along with showing failures, if any, with the documentation rendering process. Figure 3 shows the documentation homepage of GeoCAT-comp. The *NCAR* and *PYPI* badges in the *Package* row shows and links to the latest versions of the software tool distributed through NCAR's Conda channel and PyPI, respectively. The *LICENSE* badge provides a link to our software licenses, Apache License version 2.0 [Apa04], for all of the GeoCAT stack, enabling the redistribution of the open-source software products on an "as is" basis. Finally, to provide reproducibility of our software products (either for the latest or any older version), we publish version-specific Digital Object Identifiers (DOIs), which can be accessed through the *DOI* badge. This allows the end-user to accurately cite the specific version of the GeoCAT tools they used for science or research purposes.

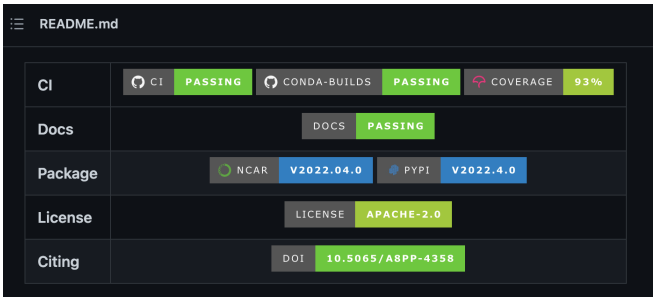


Fig. 2: *GeoCAT-comp's badges in the beginning of its README file (i.e. the home page of the Githug repository) [geob]*

GeoCAT-comp (and GeoCAT-f2py)

GeoCAT-comp is the computational component of the GeoCAT project as can be seen in Figure 4. GeoCAT-comp houses implementations of geoscience data analysis functions. Novel research and development is conducted for analyzing both structured and unstructured grid data from various research fields such as climate, weather, atmosphere, ocean, among others. In addition, some of the functionalities of GeoCAT-comp are inspired or reimplemented from the NCL in order to address the first goal of the "Pivot to Python effort. For that purpose, 114 NCL routines were selected, excluding some functionalities such as date routines, which could be handled by other packages in the Python ecosystem today. These functions were ranked by order of website documentation access from most to least, and prioritization was made based on those ranks. Today, GeoCAT-comp provides the

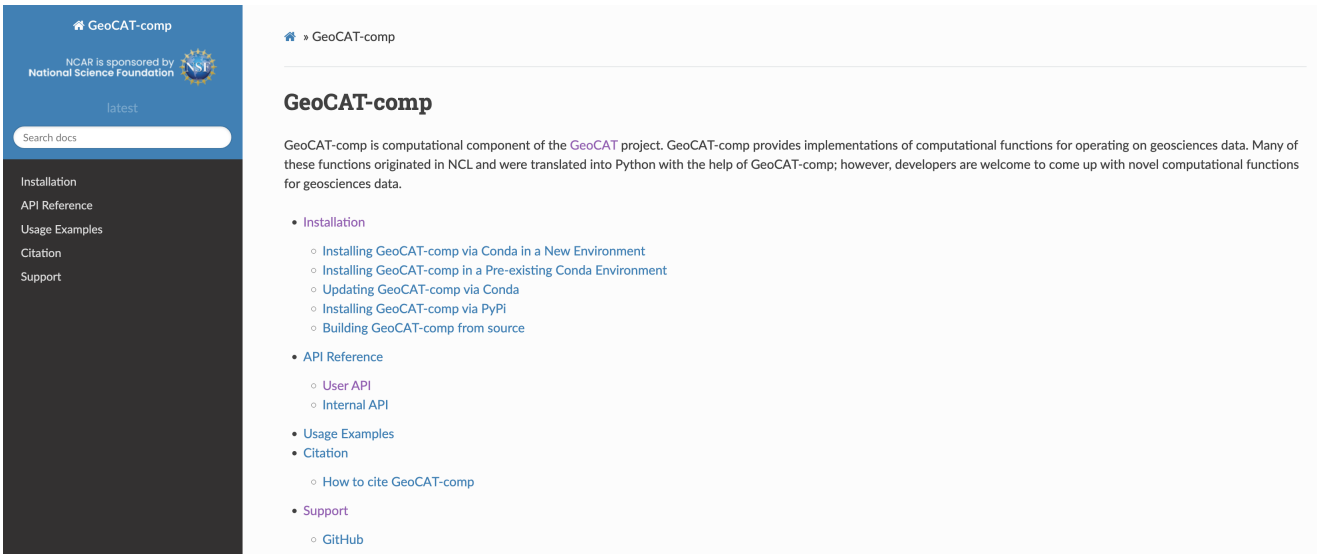


Fig. 3: *GeoCAT-comp documentation homepage built with Sphinx using a theme provided by ReadTheDocs [geoa]*

same or similar capabilities of about 39% (44 out of 114) of those functions.

Some of the functions that are made available through GeoCAT-comp are listed below, for which the GeoCAT-comp documentation [geoa] provides signatures and descriptions as well as links to the usage examples:

- Spherical harmonics (both decomposition and recomposition as well as area weighting)
- Fourier transforms such as band-block, band-pass, low-pass, and high-pass
- Meteorological variable computations such as relative humidity, dew-point temperature, heat index, saturation vapor pressure, and more
- Climatology functions such as climate average over multiple years, daily/monthly/seasonal averages, as well as anomalies
- Regridding of curvilinear grid to rectilinear grid, unstructured grid to rectilinear grid, curvilinear grid to unstructured grid, and vice versa
- Interpolation methods such as bilinear interpolation of a rectilinear to another rectilinear grid, hybrid-sigma levels to isobaric levels, and sigma to hybrid coordinates
- Empirical orthogonal function (EOF) analysis

Many of the computational functions in GeoCAT are implemented in pure Python. However, there are others that were originally implemented in Fortran but are now wrapped up in Python with the help of Numpy's F2PY, Fortran to Python interface generator. This is mostly because re-implementing some functions would require understanding of complicated algorithm flows and implementation of extensive unit tests that would end up taking too much time, compared to wrapping their already-implemented Fortran routines up in Python. Furthermore, outside contributors from science background would keep considering to add new functions to GeoCAT from their older Fortran routines in the future. To facilitate contribution, the whole GeoCAT-comp structure is split into two repositories with respect to being either pure-Python or Python with compiled code (i.e. Fortran) implementations. Such implementation layers are handled with the GeoCAT-comp and GeoCAT-f2py repositories, respectively.

GeoCAT-comp code-base does not explicitly contain or require any compiled code, making it more accessible to the general Python community at large. In addition, GeoCAT-f2py is automatically installed through GeoCAT-comp installation, and all functions contained in the "geocat.f2py" package are imported transparently into the "geocat.comp" namespace. Thus, GeoCAT-comp serves as a user API to access the entire computational toolkit even though its GitHub repository itself only contains pure Python code from the developer's perspective. Whenever prospective contributors want to contribute computational functionality in pure Python, GeoCAT-comp is the only GitHub repository they need to deal with. Therefore, there is no onus on contributors of pure Python code to build, compile, or test any compiled code (e.g. Fortran) at GeoCAT-comp level.

GeoCAT-examples (and GeoCAT-viz)

GeoCAT-examples [geoe] was created to address a few of the original milestones of NCAR's "Pivot to Python" initiative: (1) to provide the geoscience community with well-documented visualization examples for several plotting classes in the SPE, and (2) to help transition NCL users into the Python ecosystem through providing such resources. It was born in early 2020 as the result of a multi-day hackathon event among the GeoCAT team and several other scientists and developers from various NCAR labs/groups. It has since grown to house novel visualization examples and showcase the capabilities of other GeoCAT components, like GeoCAT-comp, along with newer technologies like interactive plotting notebooks. Figure 5 illustrates one of the unique GeoCAT-examples cases that was aimed at exploring the best practices for data visualization like choosing color blind friendly colormaps.

The GeoCAT-examples [geod] gallery contains over 140 example Python plotting scripts, demonstrating functionalities from Python packages like Matplotlib, Cartopy, Numpy, and Xarray. The gallery includes plots from a range of visualization categories such as box plots, contours, meteograms, overlays, projections, shapefiles, streamlines, and trajectories among others. The plotting categories and scripts under GeoCAT-examples cover almost all of the NCL plot types and techniques. In addition, GeoCAT-examples houses plotting examples for individual GeoCAT-comp analysis functions.

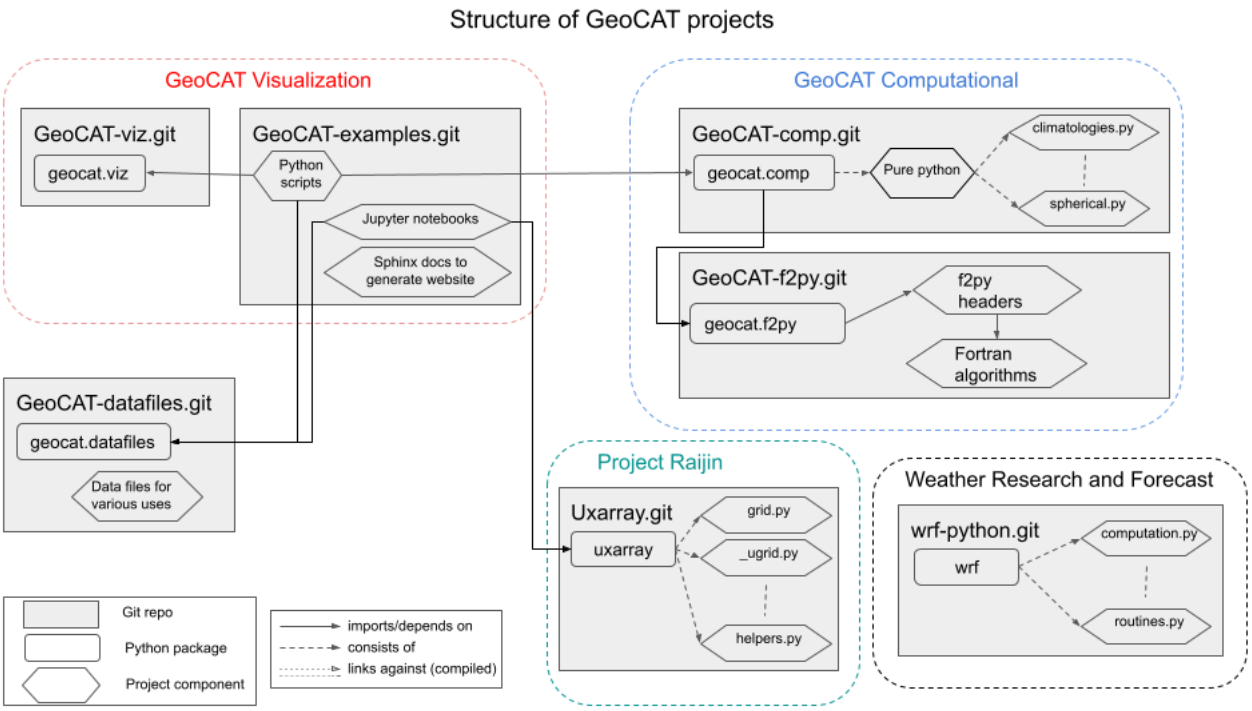


Fig. 4: *GeoCAT project structure with all of the software tools [geoc]*

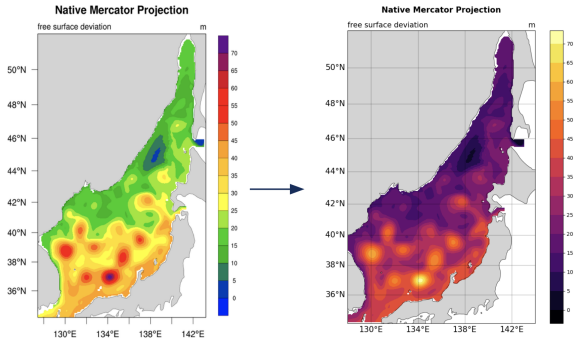


Fig. 5: *Comparison between NCL (left) and Python (right) when choosing a colormap; GeoCAT-examples aiming at choosing color blind friendly colormaps [SEKZ22]*

Despite Matplotlib and Cartopy's capabilities to reproduce almost all of NCL plots, there was one significant caveat with using their low-level implementations against NCL: NCL's high-level plotting functions allowed scientists to plot most of the cases in only tens of lines of codes (LOC) while the Matplotlib and Cartopy stack required writing a few hundred LOC. In order to build a higher-level implementation on top of Matplotlib and Cartopy while recreating the NCL-like plots (from vital plotting capabilities that were not readily available in the Python ecosystem at the time such as Taylor diagrams and curly vectors to more stylistic changes such as font sizes, color schemes, etc. that resemble NCL plots), the GeoCAT-viz library [geof] was implemented. Use of functions from this library in GeoCAT-examples significantly reduces the LOC requirements for most of the visualization examples to comparable numbers to those of NCL's. Figure 6 shows Taylor diagram and curly vector examples that have been created with the help of GeoCAT-viz. To exemplify how GeoCAT-viz helps keep the LOC comparable to NCL, one of the Taylor diagrams (i.e. Taylor_6) took 80 LOC in NCL, and its Python implementation in GeoCAT-examples takes 72 LOC. If many of the Matplotlib functions (e.g. figure and axes initialization, adjustment of several axes parameters, call to plotting functions for Taylor diagram, management of grids, addition of titles, contours, etc.) used in this example weren't wrapped up in GeoCAT-viz [geof], the same visualization would easily end up in around two hundred LOC.

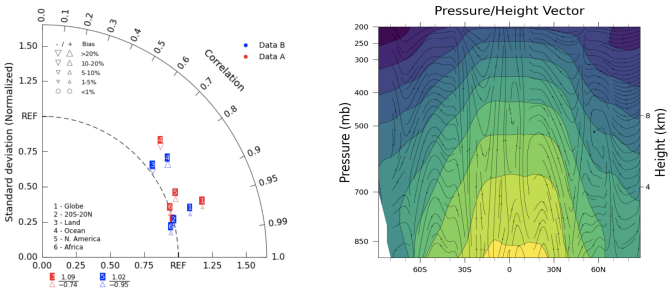


Fig. 6: *Taylor diagram and curly vector examples that created with the help of GeoCAT-viz*

Recently, the GeoCAT team has been focused on interactive plotting technologies, especially for larger data sets that contain millions of data points. This effort was centered on unstructured grid visualization as part of Project Raijin, which is detailed in a later section in this manuscript. That is because unstructured meshes are a great research and application field for big data and interactivity such as zoom in/out for regions of interest. As a result of this effort, we created a new notebooks gallery under GeoCAT-examples to house such interactive data visualizations. The first notebook, a screenshot from which is shown in Figure 7, in this gallery is implemented via the Datashader and Holoviews

ecosystem [Anaa], and it provides a high-performance, interactive visualization of a Model for Prediction Across Scales (MPAS) Global Storm-Resolving Model weather simulation dataset. The interactivity features are pan and zoom to reveal greater data fidelity globally and regionally. The data used in this work is the courtesy of the DYAMOND effort [SSA+19] and has varying resolutions from 30 km to 3.75 km. Our notebook in the gallery uses the 30 km resolution data for the users to be able to download and work on it in their local configuration. However, our work with the 3.75 km resolution data (i.e. about 42 million hexagonal cells globally) showed that rendering the data took only a few minutes on a decent laptop, even without any parallelization. The main reason behind such a high performance was that we used the cell-to-node connectivity information in the MPAS data to render the native grid directly (i.e. without remapping to the structured grid) along with utilizing the Datashader stack. Without using the connectivity information, it would require to run much costly Delaunay triangulation. The notebook provides a comparison between these two approaches as well.

GeoCAT-datafiles

GeoCAT-datafiles is GeoCAT's small data storage component as a Github repository. This tool houses many datasets in different file formats such as NetCDF, which can be used along with other GeoCAT tools or ad-hoc data needs in any other Python script. The datasets can be accessed by the end-user through a lightweight convenience function:

```
geocat.datafiles.get("folder_name/filename")
```

GeoCAT-datafiles fetches the file by simply reading from the local storage, if any, or downloading from the GeoCAT-datafiles repository, if not in the local storage, with the help of Pooch framework [USR+20].

WRF-Python

WRF-Python was created in early 2017 in order to replicate NCL's Weather Research and Forecasting (WRF) package in the SPE, and it covers 100% of the routines in that package. About two years later, NCAR's "Pivot to Python" initiative was announced, and the GeoCAT team has taken over development and maintenance of WRF-Python.

The package focuses on creating a Python package that eliminates the need to work across multiple software platforms when using WRF datasets. It contains more than 30 computational (e.g. diagnostic calculations, several interpolation routines) and visualization routines that aim at reducing the amount of post-processing tools necessary to visualize WRF output files.

Even though there is no continuous development in WRF-Python, as is seen in the rest of the GeoCAT stack, the package is still maintained with timely responses and bug-fix releases to the issues reported by the user community.

Project Raijin

"Collaborative Research: EarthCube Capabilities: Raijin: Community Geoscience Analysis Tools for Unstructured Mesh Data", i.e. Project Raijin, of the consortium between NCAR and Pennsylvania State University has been awarded by NSF 21-515 EarthCube for an award period of 1 September, 2021 - 31 August, 2024 [NSF21]. Project Raijin aims at developing community-owned, sustainable, scalable tools that facilitate operating on unstructured climate and global weather data [rai]. The GeoCAT team is in charge of the software development of Project Raijin, which mainly consists of implementing visualization and analysis functions in the SPE to be executed on native grids. While doing so, GeoCAT is also responsible for establishing an open development environment, clearly documenting the implementation work, and aligning deployments with the project milestones as well as SPE requirements and specifications.

GeoCAT has created the Xarray-based Uxarray package [uxa] to recognize unstructured grid models through partnership with geoscience community groups. UXarray is built on top of the built-in Xarray Dataset functionalities while recognizing several unstructured grid formats (UGRID, SCRIP, and Exodus for now). Since there are more unstructured mesh models in the community than UXarray natively supports, its architecture will also support addition of new models. Figure 8 shows the regularly structured "latitude-longitude" grids versus a few unstructured grid models.

The UXarray project has implemented data input/output functions for UGRID, SCRIP, and Exodus, as well as methods for surface area and integration calculations so far. The team is currently conducting open discussions (through GitHub Discussions) with community members, who are interested in unstructured grids research and development in order to prioritize data analysis operators to be implemented throughout the project lifecycle.

Scalability

GeoCAT is aware of the fact that today's geoscientific models are capable of generating huge sizes of data. Furthermore, these datasets, such as those produced by global convective-permitting models, are going to grow even larger in size in the future. Therefore, computational and visualization functions that are being developed in the geoscientific research and development workflows need to be scalable from personal devices (e.g. laptops) to HPC (e.g. NCAR's Casper, Cheyenne, and upcoming Derecho clusters) and cloud platforms (e.g. AWS).

In order to keep up with the scalability objectives, GeoCAT functions are implemented to operate on Dask arrays in addition to natively supporting NumPy arrays and Xarray DataArrays. Therefore, the GeoCAT functions can trivially and transparently be parallelized to be run on shared-memory and distributed-memory platforms after having Dask cluster/client properly configured and functions fed with Dask arrays or Dask-backed Xarray DataArrays (i.e. chunked Xarray DataArrays that wrap up Dask arrays).

Open Development

To ensure community involvement at every level in the development lifecycle, GeoCAT is committed to an open development model. In order to implement this model, GeoCAT provides all of its software tools as GitHub repositories with public GitHub project boards and roadmaps, issue tracking and development reviewing, comprehensive documentation for users and contributors such as Contributor's Guide [geoc] and toolkit-specific documentation, along with community announcements on the GeoCAT blog. Furthermore, GeoCAT encourages community feedback and contribution at any level with inclusive and welcoming language. As a result of this, community requests and feedback have played significant role in forming and revising the GeoCAT roadmap and projects' scope.

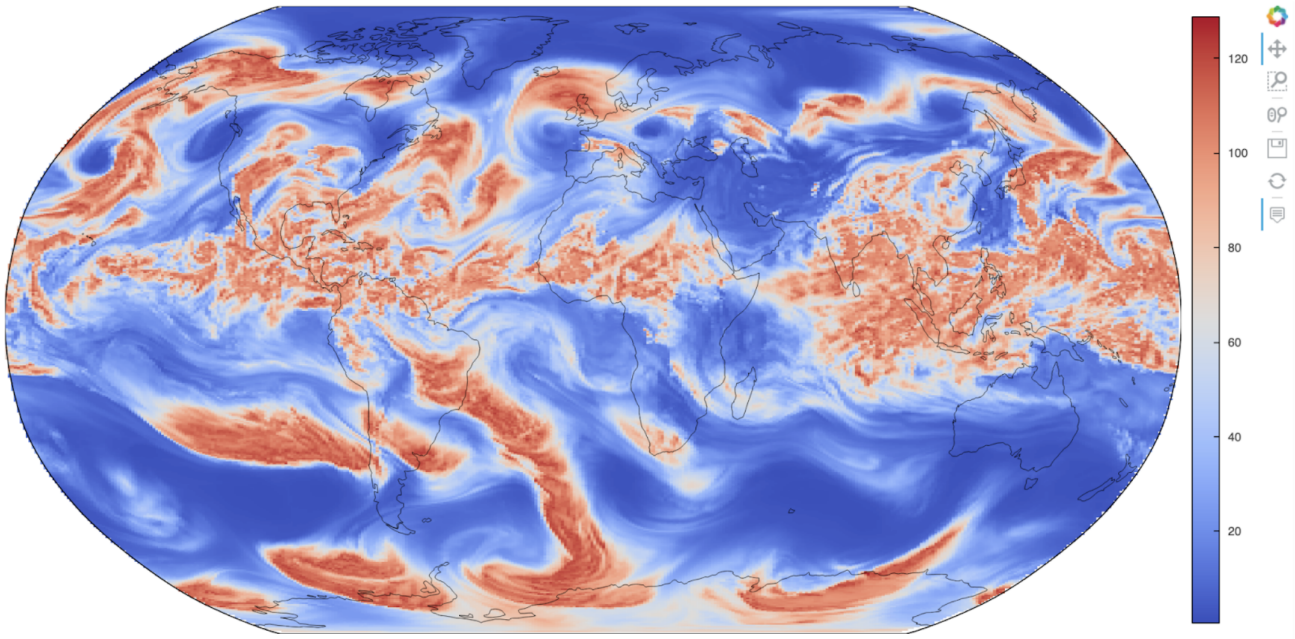


Fig. 7: *The interactive plot interface from the MPAS visualization notebook in GeoCAT-examples*

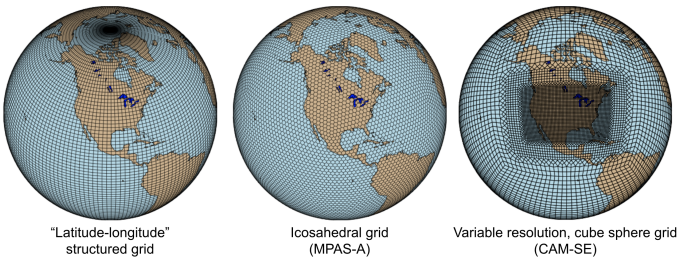


Fig. 8: *Regular grid (left) vs MPAS-A & CAM-SE grids*

Community engagement

To further promote engagement with the geoscience community, GeoCAT organizes and attends various community events. First of all, scientific conferences and meetings are great venues for such a scientific software engineering project to share updates and progress with the community. For instance, the American Meteorological Society (AMS) Annual Meeting and American Geophysical Union (AGU) Fall Meeting are two significant scientific events that the GeoCAT team presented one or multiple publications every year since its birth to inform the community. The annual Scientific Computing with Python (SciPy) conference is another great fit to showcase what GeoCAT has been conducting in geoscience. The team also attended The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC) a few times to keep up-to-date with the industry state-of-the-arts in these technologies.

Creating internship projects is another way of improving community interactions as it triggers collaboration through GeoCAT, institutions, students, and university in general. The GeoCAT team, thus,encourages undergraduate and graduate student engagement in the Python ecosystem through participation in NCAR's Summer Internships in Parallel Computational Science (SIParCS). Such programs are quite beneficial for both students and scientific software development teams. To exemplify, GeoCAT-examples and GeoCAT-viz in particular has received significant contributions through SIParCS in 2020 and 2021 summers (i.e. tens of visualization examples as well as important infrastructural changes were made available by our interns) [CKZ+22] [LLZ+21] [CFS21]. Furthermore, the team has created three essential and one collaboration project through SIParCS 2022 summer through which advanced geoscientific visualization, unstructured grid visualization and data analysis, Fortran to Python algorithm and code development, as well as GPU optimization for GeoCAT-comp routines will be investigated.

Project Pythia

The GeoCAT effort is also a part of the NSF funded Project Pythia. Project Pythia aims to provide a public, web-accessible training resource that could help educate earth scientists to more effectively use the SPE and cloud computing for dealing with big data in geosciences. GeoCAT helps with Pythia development through content creation and infrastructure contributions. GeoCAT has also contributed several Python tutorials (such as Numpy, Matplotlib, Cartopy, etc.) to the educational resources created through Project Pythia. These materials consist of live tutorial sessions, interactive Jupyter notebook demonstrations, Q&A sessions, as well as published video recording of the event on Pythia's Youtube channel. As a result, it helps us engage with the community through multiple channels.

Future directions

GeoCAT aims to keep increasing the number of data analysis and visualization functionalities in both structured and unstructured meshes with the same pace as has been done so far. The team will continue prioritizing scalability and open development in future development and maintenance of its software tools landscape. To achieve the goals with scalability of our tools, we will ensure our implementations are compatible with the state-of-the-art and up-to-date with the best practices of the technology we are using, e.g.

Dask. To enhance the community involvement in our open development model, we will continue interacting with the community members through significant events such as Pangeo community meetings, scientific conferences, tutorials and workshops of GeoCAT's own as well as other community members; we will keep our timely communication with the stakeholders through GitHub assets and other communication channels.

REFERENCES

[Anaa] Anaconda. Datashader. https://datashader.org/. Online; accessed 29 June 2022.

[Anab] Anaconda, Inc. Conda package manager. https://docs.conda.io/en/latest/. Online; accessed 18 May 2022.

[Apa04] Apache Software Foundation. Apache License, version 2.0. https://www.apache.org/licenses/LICENSE-2.0, 2004. Online; accessed 18 May 2022.

[BBHH12] David Brown, Rick Brownrigg, Mary Haley, and Wei Huang. NCAR Command Language (ncl), 2012. doi:http://dx.doi.org/10.5065/D6WD3XH5.

[Bra21] Georg Brandl. Sphinx documentation. *URL http://sphinx-doc.org/sphinx. pdf*, 2021.

[CEMZ21] John Clyne, Orhan Eroglu, Brian Medeiros, and Colin M Zarzycki. Project raijin: Community geoscience analysis tools for unstructured grids. In *AGU Fall Meeting 2021*. AGU, 2021.

[CFS21] Heather Rose Craker, Claire Anne Fiorino, and Michaela Victoria Sizemore. Rebuilding the ncl visualization gallery in python. In *101nd American Meteorological Society Annual Meeting*. AMS, 2021.

[CKZ+22] Heather Craker, Alea Kootz, Anissa Zacharias, Michaela Sizemore, and Orhan Eroglu. NCAR's GeoCAT Announcement of Computational Tools. In *102nd American Meteorological Society Annual Meeting*. AMS, 2022.

[cod] Codecov. https://about.codecov.io/. Online; accessed 18 May 2022.

[geoa] GeoCAT-comp documentation page. https://geocat-comp.readthedocs.io/en/latest/index.html. Online; accessed 20 May 2022. doi:doi:10.5281/zenodo.6607205.

[geob] GeoCAT-comp GitHub repository. https://github.com/NCAR/geocat-comp. Online; accessed 20 May 2022. doi:doi:10.5281/zenodo.6607205.

[geoc] GeoCAT Contributor's Guide. https://geocat.ucar.edu/pages/contributing.html. Online; accessed 20 May 2022. doi:10.5065/a8pp-4358.

[geod] GeoCAT-examples documentation page. https://geocat-examples.readthedocs.io/en/latest/index.html. Online; accessed 20 May 2022. doi:10.5281/zenodo.6678258.

[geoe] GeoCAT-examples GitHub repository. https://github.com/NCAR/geocat-examples. Online; accessed 20 May 2022. doi:10.5281/zenodo.6678258.

[geof] GeoCAT-viz GitHub repository. https://github.com/NCAR/geocat-viz. Online; accessed 20 May 2022. doi:10.5281/zenodo.6678345.

[Geo19] GeoCAT. The future of NCL and the Pivot to Python. https://www.ncl.ucar.edu/Document/Pivot_to_Python, 2019. Online; accessed 17 May 2022. doi:http://dx.doi.org/10.5065/D6WD3XH5.

[Git] GitHub. Github Actions. https://docs.github.com/en/actions. Online; accessed 18 May 2022.

[HH17] Stephan Hoyer and Joseph Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1):10, 2017. doi:http://doi.org/10.5334/jors.148.

[HRA18] Joseph Hamman, Matthew Rocklin, and Ryan Abernathy. Pangeo: A big-data ecosystem for scalable earth system science. EGU General Assembly Conference Abstracts, 2018.

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

[LLZ+21] Erin Lincoln, Jiaqi Li, Anissa Zacharias, Michaela Sizemore, Orhan Eroglu, and Julia Kent. Expanding and strengthening the transition from NCL to Python visualizations. In *AGU Fall Meeting 2021*. AGU, 2021.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015. doi:https://doi.org/10.1145/2833157.2833162.

[Met15] Met Office. *Cartopy: a cartographic python library with a matplotlib interface*. Exeter, Devon, 2010 - 2015. URL: http://scitools.org.uk/cartopy.

[MR15] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 126 – 132, 2015. doi:10.25080/Majora-7b98e3ed-013.

[NSF21] NSF. Collaborative research: Earthcube capabilities: Raijin: Community geoscience analysis tools for unstructured mesh data. https://nsf.gov/awardsearch/showAward?AWD_ID=2126458&HistoricalAwards=false, 2021. Online; accessed 17 May 2022.

[Pyt] Python Software Foundation. The Python Package Index - PyPI. https://pypi.org/. Online; accessed 18 May 2022.

[rai] Raijin homepage. https://raijin.ucar.edu/. Online; accessed 21 May 2022.

[rea] ReadTheDocs. https://readthedocs.org/. Online; accessed 18 May 2022.

[SEKZ22] Michaela Sizemore, Orhan Eroglu, Alea Kootz, and Anissa Zacharias. Pivoting to Python: Lessons Learned in Recreating the NCAR Command Language in Python. 102nd American Meteorological Society Annual Meeting, 2022.

[SSA+19] Bjorn Stevens, Masaki Satoh, Ludovic Auger, Joachim Biercamp, Christopher S Bretherton, Xi Chen, Peter Düben, Falko Judt, Marat Khairoutdinov, Daniel Klocke, et al. DYAMOND: the DYnamics of the Atmospheric general circulation Modeled On Non-hydrostatic Domains. *Progress in Earth and Planetary Science*, 6(1):1–17, 2019. doi:https://doi.org/10.1186/s40645-019-0304-z.

[USR+20] Leonardo Uieda, Santiago Rubén Soler, Rémi Rampin, Hugo Van Kemenade, Matthew Turk, Daniel Shapero, Anderson Banihirwe, and John Leeman. Pooch: A friend to fetch your data files. *Journal of Open Source Software*, 5(45):1943, 2020. doi:10.21105/joss.01943.

[uxa] UXarray GitHub repository. https://github.com/UXARRAY/uxarray. Online; accessed 20 May 2022. doi:10.5281/zenodo.5655065.

popmon: Analysis Package for Dataset Shift Detection

Simon Brugman[‡*], Tomas Sostak[§], Pradyot Patil[‡], Max Baak[‡]

Abstract—popmon is an open-source Python package to check the stability of a tabular dataset. popmon creates histograms of features binned in time-slices, and compares the stability of its profiles and distributions using statistical tests, both over time and with respect to a reference dataset. It works with numerical, ordinal and categorical features, on both pandas and Spark dataframes, and the histograms can be higher-dimensional, e.g. it can also track correlations between sets of features. popmon can automatically detect and alert on changes observed over time, such as trends, shifts, peaks, outliers, anomalies, changing correlations, etc., using monitoring business rules that are either static or dynamic. popmon results are presented in a self-contained report.

Index Terms—dataset shift detection, population shift, covariate shift, histogramming, profiling

Fig. 1: *The popmon package logo*

Introduction

Tracking model performance is crucial to guarantee that a model behaves as designed and trained initially, and for determining whether to promote a model with the same initial design but trained on different data to production. Model performance depends directly on the data used for training and the data predicted on. Changes in the latter (e.g. certain word frequency, user demographics, etc.) can affect the performance and make predictions unreliable.

Given that input data often change over time, it is important to track changes in both input distributions and delivered predictions periodically, and to act on them when they are significantly different from past instances – e.g. to diagnose and retrain an incorrect model in production. Predictions may be far ahead in time, so the performance can only be verified later, for example in one year. Taking action at that point might already be too late.

To make monitoring both more consistent and semi-automatic, ING Bank has created a generic Python package called popmon. popmon monitors the stability of data populations over time and detects dataset shifts, based on techniques from statistical process control and the dataset shift literature.

popmon employs so-called dynamic monitoring rules to flag and alert on changes observed over time. Using a specified reference dataset, from which observed levels of variation are extracted automatically, popmon sets allowed boundaries on the input data. If the reference dataset changes over time, the effective ranges on the input data can change accordingly. Dynamic monitoring rules make it easy to detect which (combinations of) features are most affected by changing distributions.

popmon is light-weight. For example, only one line is required to generate a stability report.

```
report = popmon.df_stability_report(
    df,
    time_axis="date",
    time_width="1w",
    time_offset="2022-1-1"
)
report.to_file("report.html")
```

The package is built on top of Python's scientific computing ecosystem (numpy, scipy [HMvdW+20], [VGO+20]) and supports pandas and Apache Spark dataframes [pdt20], [WM10], [ZXW+16]. This paper discusses how popmon monitors for dataset changes. The popmon code is modular in design and user configurable. The project is available as open-source software.[1]

Related work

Many algorithms detecting dataset shift exist that follow a similar structure [LLD+18], using various data structures and algorithms at each step [DKVY06], [QAWZ15]. However, few are readily available to use in production. popmon offers both a framework that generalizes pipelines needed to implement those algorithms, and default data drift pipelines, built on histograms with statistical comparisons and profiles (see Sec. data representation).

Other families of tools have been developed that work on individual data points, for model explanations (e.g. SHAP [LL17], feature attributions [SLL20]), rule-based data monitoring (e.g. Great Expectations, Deequ [GCSG22], [SLS+18]) and outlier detection (e.g. [RGL19], [LPO17]).

alibi-detect [KVLC+20], [VLKV+22] is somewhat similar to popmon. This is an open-source Python library that

* Corresponding author: simon.brugman@ing.com
‡ ING Analytics Wholesale Banking
§ Vinted

1. See https://github.com/ing-bank/popmon for code, documentation, tutorials and example stability reports.

focuses on outlier, adversarial and drift detection. It allows for monitoring of tabular, text, images and time series data, using both online and offline detectors. The backend is implemented in TensorFlow and PyTorch. Much of the reporting functionality, such as feature distributions, are restricted to the (commercial) enterprise version called `seldon-deploy`. Integrations for model deployment are available based on Kubernetes. The infrastructure setup thus is more complex and restrictive than for `popmon`, which can run on any developer's machine.

Contributions

The advantage of `popmon`'s dynamic monitoring rules over conventional static ones, is that little prior knowledge is required of the input data to set sensible limits on the desired level of stability. This makes `popmon` a scalable solution over multiple datasets.

To the best of our knowledge, no other monitoring tool exists that suits our criteria to monitor models in production for dataset shift. In particular, no other, light-weight, open-source package is available that performs such extensive stability tests of a pandas or Spark dataset.

We believe the combination of wide applicability, out-of-the-box performance, available statistical tests, and configurability makes `popmon` an ideal addition to the toolbox of any data scientist or machine learning engineer.

Approach

`popmon` tests the dataset stability and reports the results through a sequence of steps (Fig. 2):

1) The data are represented by histograms of features, binned in time-slices (Sec. data representation).
2) The data is arranged according to the selected reference type (Sec. comparisons).
3) The stability of the profiles and distributions of those histograms are compared using statistical tests, both with respect to a reference and over time. It works with numerical, ordinal, categorical features, and the histograms can be higher-dimensional, e.g. it can also track correlations between any two features (Sec. comparisons).
4) `popmon` can automatically flag and alert on changes observed over time, such as trends, anomalies, changing correlations, etc, using monitoring rules (Sec. alerting).
5) Results are reported to the user via a dedicated, self-contained report (Sec. reporting).

Dataset shift

In the context of supervised learning, one can distinguish dataset shift as a shift in various distributions:

1) Covariate shift: shift in the independent variables ($p(x)$).
2) Prior probability shift: shift in the target variable (the class, $p(y)$).
3) Concept shift: shift in the relationship between the independent and target variables (i.e. $p(x|y)$).

Note that there is a lot of variation in terminology used, referring to probabilities prevents this ambiguity. For more information on dataset shift see Quinonero-Candela et al. [QCSSL08].

`popmon` is primarily interested in monitoring the distributions of features $p(x)$ and labels $p(y)$ for monitoring trained classifiers. These data in deployment ideally resembles the training data.

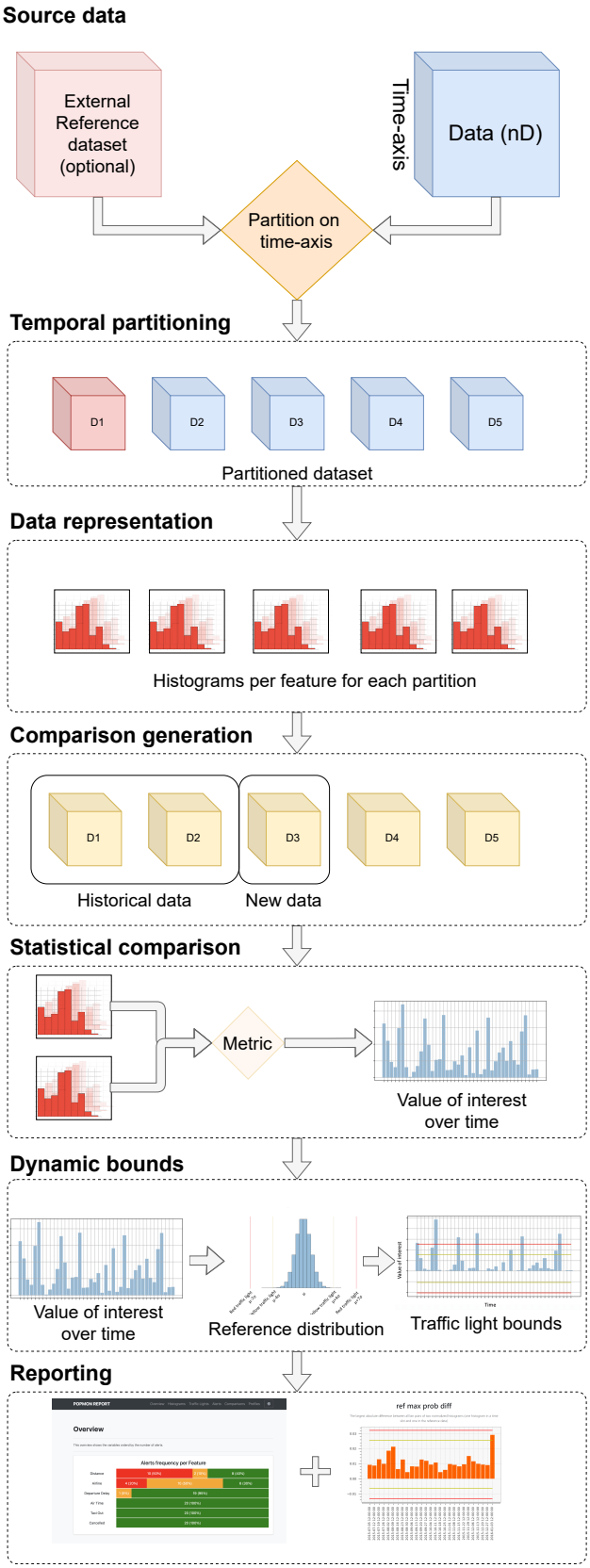


Fig. 2: *Step-by-step overview of `popmon`'s pipeline as described in section approach onward.*

However, the package can be used more widely, for instance by monitoring interactions between features and the label, or the distribution of model predictions.

Temporal representation

`popmon` requires features to be distributed as a function of time (bins), which can be provided in two ways:

1) **Time axis**. Two-dimensional (or higher) distributions are provided, where the first dimension is time and the second is the feature to monitor. To get time slices, the time column needs to be specified, e.g. "date", including the bin width, e.g. one week ("1w"), and the offset, which is the lower edge of one time-bin, e.g. a certain start date ("2022-1-1").
2) **Ordered data batches**. A set of distributions of features is provided, corresponding to a new batch of data. This batch is considered a new time-slice, and is stitched to an existing set of batches, in order of incoming batches, where each batch is assigned a unique, increasing index. Together the indices form an artificial, binned time-axis.

Data representation

`popmon` uses histogram-based monitoring to track potential dataset shift and outliers over time, as detailed in the next subsection.

In the literature, alternative data representations are also employed, such as kdq-trees [DKVY06]. Different data representations are in principle compatible with the `popmon` pipeline, as it is similarly structured to alternative methods (see [LLD+18], c.f. Fig 5).

Dimensionality reduction techniques may be used to transform the input dataset into a space where the distance between instances are more meaningful for comparison, before using `popmon`, or in-between steps. For example a linear projection may be used as a preprocessing step, by taking the principal components of PCA as in [QAWZ15]. Machine learning classifiers or autoencoders have also been used for this purpose [LWS18], [RGL19] and can be particularly helpful for high-dimensional data such as images or text.

Histogram-based monitoring

There are multiple reasons behind the histogram-based monitoring approach taken in `popmon`.

Histograms are small in size, and thus are efficiently stored and transferred, regardless of the input dataset size. Once data records have been aggregated feature-wise, with a minimum number of entries per bin, they are typically no longer privacy sensitive (e.g. knowing the number of records with age 30-35 in a dataset).

`popmon` is primarily looking for changes in data distributions. Solely monitoring the (main) profiles of a distribution, such as the mean, standard deviation and min and max values, does not necessarily capture the changes in a feature's distribution. Well-known examples of this are Anscome's Quartet [Ans73] and the dinosaurs datasets [MF17], where – between different datasets – the means and correlation between two features are identical, but the distributions are different. Histograms of the corresponding features (or feature pairs), however, do capture the corresponding changes.

Implementation

For the creation of histograms from data records the open-source `histogrammar` package has been adopted. `histogrammar` has been implemented in both Scala and Python [PS21], [PSSE16], and works on Spark and pandas dataframes respectively. The two implementations have been tested extensively to guarantee compatibility. The histograms coming out of `histogrammar` form the basis of the monitoring code in `popmon`, which otherwise does not require input dataframes. In other words, the monitoring code itself has no Spark or pandas data dependencies, keeping the code base relatively simple.

Histogram types

Three types of histograms are typically used:

- **Normal histograms**, meant for numerical features with known, fixed ranges. The bin specifications are the lowest and highest expected values and the number of (equidistant) bins.
- **Categorical histograms**, for categorical and ordinal features, typically boolean or string-based. A categorical histogram accepts any value: when not yet encountered, it creates a new bin. No bin specifications are required.
- **Sparse histograms** are open-ended histograms, for numerical features with no known range. The bin specifications only need the bin-width, and optionally the origin (the lower edge of bin zero, with a default value of zero). Sparse histograms accept any value. When the value is not yet encountered, a new bin gets created.

For normal and sparse histograms reasonable bin specifications can be derived automatically. Both categorical and sparse histograms are dictionaries with histogram properties. New (index, bin) pairs get created whenever needed. Although this could result in out-of-memory problems, e.g. when histogramming billions of unique strings, in practice this is typically not an issue, as this can be easily mitigated. Features may be transformed into a representation with a lower number of distinct values, e.g. via embedding or substrings; or one selects the top-n most frequently occurring values.

Open-ended histograms are ideal for monitoring dataset shift and outliers: they capture any kind of (large) data change. When there is a drift, there is no need to change the low- and high-range values. The same holds for outlier detection: if a new maximum or minimum value is found, it is still captured.

Dimensionality

A histogram can be multi-dimensional, and any combination of types is possible. The first dimension is always the time axis, which is always represented by a sparse histogram. The second dimension is the feature to monitor over time. When adding a third axis for another feature, the heatmap between those two features is created over time. For example, when monitoring financial transactions: the first axis could be time, the second axis client type, and the third axis transaction amount.

Usually one feature is followed over time, or at maximum two. The synthetic datasets in section synthetic datasets contain examples of higher-dimensional histograms for known interactions.

Additivity

Histograms are additive. As an example, a batch of data records arrives each week. A new batch arrives, containing timestamps that were missing in a previous batch. When histograms are made of the new batch, these can be readily summed with the histograms of the previous batches. The missing records are immediately put into the right time-slices.

It is important that the bin specifications are the same between different batches of data, otherwise their histograms cannot be summed and comparisons are impossible.

Limitations

There is one downside to using histograms: since the data get aggregated into bins, and profiles and statistical tests are obtained from the histograms, slightly lower resolution is achieved than on the full dataset. In practice, however, this is a non-issue; histograms work great for data monitoring. The reference type and time-axis binning configuration allow the user for selecting an effective resolution.

Comparisons

In `popmon` the monitoring of data stability is based on statistical process control (SPC) techniques. SPC is a standard method to manage the data quality of high-volume data processing operations, for example in a large data warehouse [Eng99]. The idea is as follows. Most features have multiple sources of variation from underlying processes. When these processes are stable, the variation of a feature over time should remain within a known set of limits. The level of variation is obtained from a reference dataset, one that is deemed stable and trustworthy.

For each feature in the input data (except the time column), the stability is determined by taking the reference dataset – for example the data on which a classification model was trained – and contrasting each time slot in the input data.

The comparison can be done in two ways:

1) **Comparisons**: statistically comparing each time slot to the reference data (for example using Kolmogorov-Smirnov testing, χ^2 testing, or the Pearson correlation).
2) **Profiles**: for example, tracking the mean of a distribution over time and contrasting this to the reference data. Similar analyses can be done for other summary statistics, such as the median, min, max or quantiles. This is related to the CUsUM technique [Pag54], a well-known method in SPC.

Reference types

Consider X to be an N-dimensional dataset representing our reference data, and X' to be our incoming data. A covariate shift occurs when $p(X) \neq p(X')$ is detected. Different choices for X and X' may detect different types of drift (e.g. sudden, gradual, incremental). $p(X)$ is referred to as the reference dataset.

Many change-detection algorithms use a window-based solution that compares a static reference to a test window [DKVY06], or a sliding window for both, where the reference is dynamically updated [QAWZ15]. A static reference is a wise choice for monitoring of a trained classifier: the performance of such a classifier depends on the similarity of the test data to the training data. Moreover, it may pick up an incremental departure (trend) from the initial distribution, that will not be significant in comparison to the adjacent time-slots. A sliding reference, on the other hand, is updated with more recent data, that incorporates this trend. Consider the case where the data contain a price field that is yearly indexed to the inflation, then using a static reference may alert purely on the trend.

The reference implementations are provided for common scenarios, such as working with a fixed dataset, batched dataset or with streaming data. For instance, a fixed dataset is common for exploratory data analysis and one-off monitoring, whereas batched or streaming data is more common in a production setting.

The reference may be static or dynamic. Four different reference types are possible:

1) **Self-reference**. Using the full dataset on which the stability report is built as a reference. This method is static: each time slot is compared to all the slots in the dataset. This is the default reference setting.
2) **External reference**. Using an external reference set, for example the training data of your classifier, to identify which time slots are deviating. This is also a static method: each time slot is compared to the full reference set.
3) **Rolling reference**. Using a rolling window on the input dataset, allowing one to compare each time slot to a window of preceding time slots. This method is dynamic: one can set the size of the window and the shift from the current time slot. By default the 10 preceding time slots are used.
4) **Expanding reference**. Using an expanding reference, allowing one to compare each time slot to all preceding time slots. This is also a dynamic method, with variable window size, since all available previous time slots are used. For example, with ten available time slots the window size is 9.

Statistical comparisons

Users may have various reasons to prefer a two-sample test over another. The appropriate comparison depends on our confidence in the reference dataset [Ric22], and certain tests may be more common in some fields. Many common tests are related [DKVY06], e.g. the χ^2 function is the first-order expansion of the KL distance function.

Therefore, `popmon` provides an extensible framework that allows users to provide custom two-sample tests using a simple syntax, via the registry pattern:

```
@Comparisons.register(key="jsd", description="JSD")
def jensen_shannon_divergence(p, q):
    m = 0.5 * (p + q)
    return (
        0.5 *
        (kl_divergence(p, m) + kl_divergence(q, m))
    )
```

Most commonly used test statistics are implemented, such as the Population-Stability-Index and the Jensen-Shannon divergence. The implementations of the χ^2 and Kolmogorov-Smirnov tests account for statistical fluctuations in both the input and reference distributions. For example, this is relevant when comparing adjacent, low-statistics time slices.

Profiles

Tracking the distribution of values of interest over time is achieved via profiles. These are functions of the input histogram. Metrics

may be defined for all dimensions (e.g. count, correlations), or for specific dimensions as in the case of 1D numerical histograms (e.g. quantiles). Extending the existing set of profiles is possible via a syntax similar as above:

```
@Profiles.register(
      key=["q5", "q50", "q95"],
      description=[
          "5% percentile",
          "50% percentile (median)",
          "95% percentile"
      ],
      dim=1,
      type="num"
)
def profile_quantiles(values, counts):
    return logic_goes_here(values, counts)
```

Denote $x_i(t)$ as the profile i of feature x at time t, for example the 5% quantile of the histogram of incoming transaction amounts in a given week. Identical bin specifications are assumed between the reference and incoming data. $\bar{x}_i$ is defined as the average of that metric on the reference data, and σ_{x_i} as the corresponding standard deviation.

The normalized residual between the incoming and reference data, also known as the "pull" or "Z-score", is given by:

$$\text{pull}_i(t) = \frac{x_i(t) - \bar{x}_i}{\sigma_{x_i}} .$$

When the underlying sources of variation are stable, and assuming the reference dataset is asymptotically large and independent from the incoming data, $\text{pull}_i(t)$ follows a normal distribution centered around zero and with unit width, $N(0,1)$, as dictated by the central limit theorem [Fis11].

In practice, the criteria for normality are hardly ever met. Typically the distribution is wider with larger tails. Yet, approximately normal behaviour is exhibited. Chebyshev's inequality [Che67] guarantees that, for a wide class of distributions, no more than $\frac{1}{k^2}$ of the distribution's values can be k or more standard deviations away from the mean. For example, a minimum of 75% (88.9%) of values must lie within two (three) standard deviations of the mean. These boundaries reoccur in Sec. dynamic monitoring rules.

Alerting

For alerting, `popmon` uses traffic-light-based monitoring rules, raising green, yellow or red alerts to the user. Green alerts signal the data are fine, yellow alerts serve as warnings of meaningful deviations, and red alerts need critical attention. These monitoring rules can be static or dynamic, as explained in this section.

Static monitoring rules

Static monitoring rules are traditional data quality rules (e.g. [RD00]). Denote $x_i(t)$ as metric i of feature x at time t, for example the number of NaNs encountered in feature x on a given day. As an example, the following traffic lights might be set on $x_i(t)$:

$$TL(x_i,t) = \begin{cases} \text{Green}, & \text{if } x_i(t) \leq 1 \\ \text{Yellow}, & \text{if } 1 < x_i(t) \leq 10 \\ \text{Red}, & \text{if } x_i(t) > 10 \end{cases}$$

The thresholds of this monitoring rule are fixed, and considered static over time. They need to be set by hand, to sensible values. This requires domain knowledge of the data and the processes that produce it. Setting these traffic light ranges is a time-costly process when covering many features and corresponding metrics.

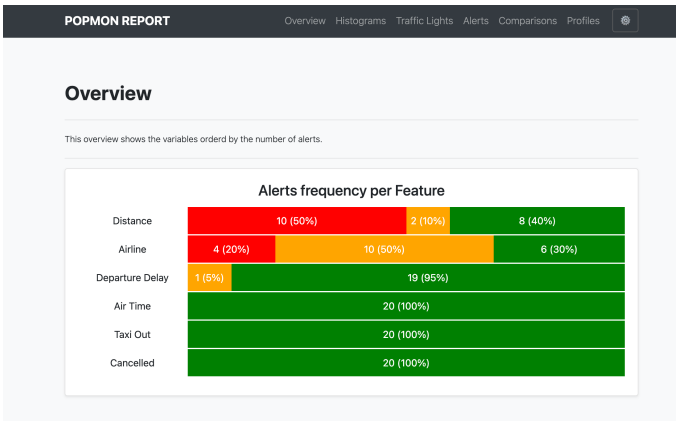


Fig. 3: *A snapshot of part of the HTML stability report. It shows the aggregated traffic light overview. This view can be used to prioritize features for inspection.*

Dynamic monitoring rules

Dynamic monitoring rules are complementary to static rules. The levels of variation in feature metrics are assumed to have been measured on the reference data. Per feature metric, incoming data are compared against the reference levels. When (significantly) outside of the known bounds, instability of the underlying sources is assumed, and a warning gets raised to the user.

`popmon`'s dynamic monitoring rules raise traffic lights to the user whenever the normalized residual $\text{pull}_i(t)$ falls outside certain, configurable ranges. By default:

$$TL(\text{pull}_i,t) = \begin{cases} \text{Green}, & \text{if } |\text{pull}_i(t)| \leq 4 \\ \text{Yellow}, & \text{if } 4 < |\text{pull}_i(t)| \leq 7 \\ \text{Red}, & \text{if } |\text{pull}_i(t)| > 7 \end{cases}$$

If the reference dataset is changing over time, the effective ranges on $x_i(t)$ can change as well. The advantage of this approach over static rules is that significant deviations in the incoming data can be flagged and alerted to the user for a large set of features and corresponding metrics, requiring little (or no) prior knowledge of the data at hand. The relevant knowledge is all extracted from the reference dataset.

With multiple feature metrics, many dynamic monitoring tests can get performed on the same dataset. This raises the multiple comparisons problem: the more inferences are made, the more likely erroneous red flags are raised. To compensate for a large number of tests being made, typically one can set wider traffic light boundaries, reducing the false positive rate.[2] The boundaries control the size of the deviations - or number of red and yellow alerts - that the user would like to be informed of.

Reporting

`popmon` outputs monitoring results as HTML stability reports. The reports offer multiple views of the data (histograms and heatmaps), the profiles and comparisons, and traffic light alerts. There are several reasons for providing self-contained reports: they can be opened in the browser, easily shared, stored as artifacts, and tracked using tools such as MLFlow. The reports also have no need for an advanced infrastructure setup, and are possible to create and

2. Alternatively one may apply the Bonferroni correction to counteract this problem [Bon36].

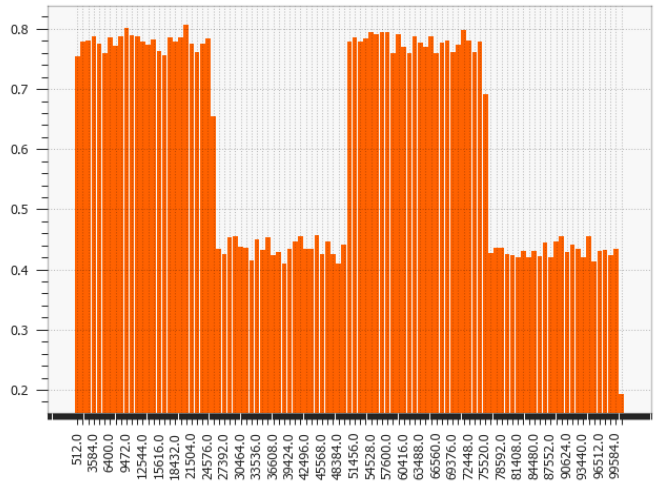


Fig. 4: LED*: Pearson correlation compared with previous histogram. The shifting points are correctly identified at every 5th of the LED dataset. Similar patterns are visible for other comparisons, e.g.* χ^2.

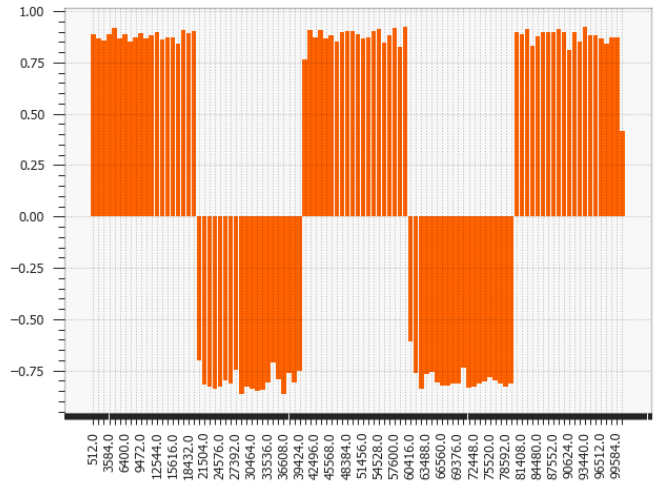


Fig. 5: Sine1*: The dataset shifts around data points 20.000, 40.000, 60.000 and 80.000 of the Sine1 dataset are clearly visible.*

view in many environments: from a local machine, a (restricted) environment, to a public cloud. If, however, a certain dashboarding tool is available, then the metrics computed by `popmon` are exposed and can be exported into that tool, for example Kibana [Ela22]. One downside of producing self-contained reports is that they can get large when the plots are pre-rendered and embedded. This is mitigated by embedding plots as JSON that are (lazily) rendered on the client-side. Plotly express [Plo22] powers the interactive embedded plots in `popmon` as of v1.0.0.

Note that multiple reference types can be used in the same stability report. For instance, `popmon`'s default reference pipelines always include a rolling comparison with window size 1, i.e. comparing to the preceding time slot.

Synthetic datasets

In the literature synthetic datasets are commonly used to test the effectiveness of dataset shift monitoring approaches [LLD+18]. One can test the detection for all kinds of shifts, as the generation process controls when and how the shift happens. `popmon` has been tested on multiple of such artificial datasets: Sine1, Sine2, Mixed, Stagger, Circles, LED, SEA and Hyperplane [PVP18], [SK], [Fan04]. These datasets cover myriad dataset shift characteristics: sudden and gradual drifts, dependency of the label on just one or multiple features, binary and multiclass labels, and containing unrelated features. The dataset descriptions and sample `popmon` configurations are available in the code repository.

The reports generated by `popmon` capture features and time bins where the dataset shift is occurring for all tested datasets. Interactions between features and the label can be used for feature selection, in addition to monitoring the individual feature distributions. The sudden and gradual drifts are clearly visible using a rolling reference, see Fig. 4 for examples. The drift in the Hyperplane dataset, incremental and gradual, is not expected to be detected using a rolling reference or self-reference. Moreover, the dataset is synthesized so that the distribution of the features and the class balance does not change [Fan04].

The process to monitor this dataset could be set up in multiple ways, one of which is described here. A logistic regression model is trained on the first 10% of the data, which is also used as static reference. The predictions of this model are added to the dataset, simulating a machine learning model in production. `popmon` is able to pick up the divergence between the predictions and the class label, as depicted in Figure 6.

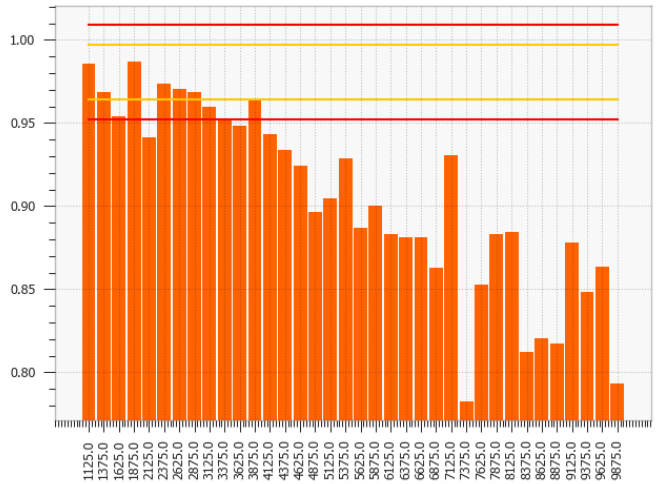


Fig. 6: Hyperplane*: The incremental drift compared to the reference dataset is observed for the PhiK correlation between the predictions and the label.*

Conclusion

This paper has presented `popmon`, an open-source Python package to check the stability of a tabular dataset. Built around histogram-based monitoring, it runs on a dataset of arbitrary size, supporting both pandas and Spark dataframes. Using the variations observed in a reference dataset, `popmon` can automatically detect and flag deviations in incoming data, requiring little prior domain knowledge. As such, `popmon` is a scalable solution that can be applied to many datasets. By default its findings get presented in a single HTML report. This makes `popmon` ideal for both exploratory data analysis and as a monitoring tool for machine learning models running in production. We believe the combination of out-of-the-box performance and presented features makes `popmon` an excellent addition to the data practitioner's toolbox.

Acknowledgements

We thank our colleagues from the ING Analytics Wholesale Banking team for fruitful discussions, all past contributors to `popmon`, and in particular Fabian Jansen and Ilan Fridman Rojas for carefully reading the manuscript. This work is supported by ING Bank.

REFERENCES

[Ans73] F.J. Anscome. Graphs in statistical analysis. *American Statistician. 27 (1)*, pages 17–21, 1973. URL: https://doi.org/10.2307/2682899, doi:10.2307/2682899.

[Bon36] Carlo Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, 8:3–62, 1936.

[Che67] Pafnutii Lvovich Chebyshev. Des valeurs moyennes, liouville's. *J. Math. Pures Appl.*, 12:177–184, 1867.

[DKVY06] Tamraparni Dasu, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *In Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*. Citeseer, 2006.

[Ela22] Elastic. Kibana, 2022. URL: https://github.com/elastic/kibana.

[Eng99] Larry English. *Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits*. Wiley, 1999.

[Fan04] Wei Fan. Systematic data selection to mine concept-drifting data streams. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, page 128–137, New York, NY, USA, 2004. Association for Computing Machinery. URL: https://doi.org/10.1145/1014052.1014069, doi:10.1145/1014052.1014069.

[Fis11] Hans Fischer. *The Central Limit Theorem from Laplace to Cauchy: Changes in Stochastic Objectives and in Analytical Methods*, pages 17–74. Springer New York, New York, NY, 2011. URL: https://doi.org/10.1007/978-0-387-87857-7_2, doi:10.1007/978-0-387-87857-7_2.

[GCSG22] Abe Gong, James Campbell, Superconductive, and Great Expectations. Great Expectations, 2022. URL: https://github.com/great-expectations/great_expectations, doi:10.5281/zenodo.5683574.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[KVLC+20] Janis Klaise, Arnaud Van Looveren, Clive Cox, Giovanni Vacanti, and Alexandru Coca. Monitoring and explainability of models in production. *arXiv preprint arXiv:2007.06299*, 2020. URL: https://doi.org/10.48550/arXiv.2007.06299, doi:10.48550/arXiv.2007.06299.

[LL17] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[LLD+18] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018. doi:10.1109/TKDE.2018.2876857.

[LPO17] David Lopez-Paz and Maxime Oquab. Revisiting classifier two-sample tests. In *International Conference on Learning Representations*, 2017.

[LWS18] Zachary Lipton, Yu-Xiang Wang, and Alexander Smola. Detecting and correcting for label shift with black box predictors. In *International conference on machine learning*, pages 3122–3130. PMLR, 2018.

[MF17] Justin Matejka and George Fitzmaurice. Same stats, different graphs: generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 1290–1294, 2017. URL: https://doi.org/10.1145/3025453.3025912, doi:10.1145/3025453.3025912.

[Pag54] Ewas S Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954. URL: https://doi.org/10.2307/2333009, doi:10.2307/2333009.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[Plo22] Plotly Development Team. Plotly.py: The interactive graphing library for Python (includes Plotly Express), 6 2022. URL: https://github.com/plotly/plotly.py.

[PS21] Jim Pivarski and Alexey Svyatkovskiy. histogrammar/histogrammar-scala: v1.0.20, April 2021. URL: https://doi.org/10.5281/zenodo.4660177, doi:10.5281/zenodo.4660177.

[PSSE16] Jim Pivarski, Alexey Svyatkovskiy, Ferdinand Schenck, and Bill Engels. histogrammar-python: 1.0.0, September 2016. URL: https://doi.org/10.5281/zenodo.61418, doi:10.5281/zenodo.61418.

[PVP18] Ali Pesaranghader, Herna Viktor, and Eric Paquet. Reservoir of diverse adaptive learners and stacking fast hoeffding drift detection methods for evolving data streams. *Machine Learning*, 107(11):1711–1743, 2018. URL: https://doi.org/10.1007/s10994-018-5719-z, doi:10.1007/s10994-018-5719-z.

[QAWZ15] Abdulhakim A Qahtan, Basma Alharbi, Suojin Wang, and Xiangliang Zhang. A pca-based change detection framework for multidimensional data streams: Change detection in multidimensional data streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–944, 2015. doi:10.1145/2783258.2783359.

[QCSSL08] Joaquin Quiñonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. *Dataset shift in machine learning*. Mit Press, 2008.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[RGL19] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. Failing loudly: An empirical study of methods for detecting dataset shift. *Advances in Neural Information Processing Systems*, 32, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/846c260d715e5b854ffad5f70a516c88-Abstract.html.

[Ric22] Oliver E Richardson. Loss as the inconsistency of a probabilistic dependency graph: Choose your model, not your loss function. In *International Conference on Artificial Intelligence and Statistics*, pages 2706–2735. PMLR, 2022.

[SK] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, page 377–382, New York, NY, USA. Association for Computing Machinery. URL: https://doi.org/10.1145/502512.502568, doi:10.1145/502512.502568.

[SLL20] Pascal Sturmfels, Scott Lundberg, and Su-In Lee. Visualizing the impact of feature attribution baselines. *Distill*, 2020. https://distill.pub/2020/attribution-baselines. doi:10.23915/distill.00022.

[SLS+18] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proc. VLDB Endow.*, 11(12):1781–1794, aug 2018. URL: https://doi.org/10.14778/3229863.3229867, doi:10.14778/3229863.3229867.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy

1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[VLKV+22] Arnaud Van Looveren, Janis Klaise, Giovanni Vacanti, Oliver Cobb, Ashley Scillitoe, and Robert Samoilescu. Alibi Detect: Algorithms for outlier, adversarial and drift detection, 4 2022. URL: https://github.com/SeldonIO/alibi-detect.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi:10.25080/Majora-92bf1922-00a.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016. URL: https://doi.org/10.1145/2934664, doi:10.1145/2934664.

pyDAMPF: a Python package for modeling mechanical properties of hygroscopic materials under interaction with a nanoprobe

Willy Menacho[‡§], Gonzalo Marcelo Ramírez-Ávila[‡§], Horacio V. Guzman[¶‖‡§*]

Abstract—pyDAMPF is a tool oriented to the Atomic Force Microscopy (AFM) community, which allows the simulation of the physical properties of materials under variable relative humidity (RH). In particular, pyDAMPF is mainly focused on the mechanical properties of polymeric hygroscopic nanofibers that play an essential role in designing tissue scaffolds for implants and filtering devices. Those mechanical properties have been mostly studied from a very coarse perspective reaching a micrometer scale. However, at the nanoscale, the mechanical response of polymeric fibers becomes cumbersome due to both experimental and theoretical limitations. For example, the response of polymeric fibers to RH demands advanced models that consider sub-nanometric changes in the local structure of each single polymer chain. From an experimental viewpoint, choosing the optimal cantilevers to scan the fibers under variable RH is not trivial.

In this article, we show how to use pyDAMPF to choose one optimal nanoprobe for planned experiments with a hygroscopic polymer. Along these lines, We show how to evaluate common and non-trivial operational parameters from an AFM cantilever of different manufacturers. Our results show in a stepwise approach the most relevant parameters to compare the cantilevers based on a non-invasive criterion of measurements. The computing engine is written in Fortran, and wrapped into Python. This aims to reuse physics code without losing interoperability with high-level packages. We have also introduced an in-house and transparent method for allowing multi-thread computations to the users of the pyDAMPF code, which we benchmarked for various computing architectures (PC, Google Colab and an HPC facility) and results in very favorable speed-up compared to former AFM simulators.

Index Terms—Materials science, Nanomechanical properties, AFM, f2py, multi-threading CPUs, numerical simulations, polymers

Introduction and Motivation

This article provides an overview of pyDAMPF, which is a BSD licensed, Python and Fortran modeling tool that enables AFM users to simulate the interaction between a probe (cantilever) and materials at the nanoscale under diverse environments. The code is packaged in a bundle and hosted on GitHub at (https://github.com/govarguz/pyDAMPF).

‡ *Instituto de Investigaciones Físicas.*
§ *Carrera de Física, Universidad Mayor de San Andrés. Campus Universitario Cota Cota. La Paz, Bolivia*
∗ *Corresponding author: horacio.guzman@ijs.si*
¶ *Department of Theoretical Physics*
‖ *Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia*

Despite the recent open-source availability of dynamic AFM simulation packages [GGG15], [MHR08], a broad usage for the assessment and planning of experiments has yet to come. One of the problems is that it is often hard to simulate several operational parameters at once. For example, most scientists evaluate different AFM cantilevers before starting new experiments. A typical evaluation criterion is the maximum exerted force that prevents invasivity of the nanoprobe into the sample. The variety of AFM cantilevers depends on the geometrical and material characteristics used for its fabrication. Moreover, manufacturers' nanofabrication techniques may change from time to time, according to the necessities of the experiments, like sharper tips and/or higher oscillation frequencies. From a simulation perspective, evaluating observables for reaching optimal results on upcoming experiments is nowadays possible for *tens* or *hundreds* of cantilevers. On top of other operational parameters in the case of dynamic AFM like the oscillation amplitude A_0, set-point A_{sp}, among other materials expected properties that may feed simulations and create simulations batches of easily *thousands* of cases. Given this context, we focus this article on choosing a cantilever out of an initial pyDAMPF database of 30. In fact, many of them are similar in terms of spring constant k_c, cantilever volume V_c and also Tip's radius R_T. Then we focus on seven archetypical and distinct cases/cantilevers to understand the characteristics of each of the parameters specified in the manufacturers' datasheets, by evaluating the maximum (peak) forces.

We present four scenarios comparing a total of seven cantilevers and the same sample, where we use as a test-case Poly-Vinyl Acetate (PVA) fiber. The first scenario (Figure 1) illustrates the difference between air and a moist environment. On the second one, a cantilever, only very soft and stiff cantilever spring constants are compared (see Figure :ref:fig1b‘). At the same time, the different volumes along the 30 cantilevers are depicted in Figure 3. A final and mostly very common comparison is scenario 4, by comparing one of the most sensitive parameters to the force of the tip's radii (see Figure 4).

The quantitative analysis for these four scenarios is presented and also the advantages of computing several simulation cases at once with our in-house development. Such a comparison is performed under the most common computers used in science, namely, personal computers (PC), cloud (Colab) and supercomputing (small Xeon based cluster). We reach a Speed-up of 20 over the former implementation [GGG15].

Another novelty of pyDAMPF is the detailed [GS05] calculation of the environmental-related parameters, like the quality factor Q.

Here, we summarize the main features of pyDAMPF are:

- Highly efficient structure in terms of time-to-result, at least one order of magnitude faster than existing approaches.
- Easy to use for scientists without a computing background, in particular in the use of multi-threads.
- It supports the addition of further AFM cantilevers and parameters into the code database.
- Allows an interactive analysis, including a graphical and table-based comparison of results through Jupyter Notebooks.

The results presented in this article are available as Google Colaboratory notebook, which facilitates to explore pyDAMPF and these examples.

Methods

Processing inputs

pyDAMPF counts with an initial database of 30 cantilevers, which can be extended at any time by accessing to the file *cantilevers_data.txt* then, the program inputs_processor.py reads the cantilever database and asks for further physical and operational variables, required to start the simulations. This will generate *tempall.txt*, which contains all cases *e.g.* 30 to be simulated with pyDAMPF

```
def inputs_processor(variables,data):
    a,b = np.shape(data)
    final = gran_permutador( variables, data)
    f_name = ' tempall.txt'
    np.savetxt(f_name,final)
    directory = os.getcwd()
    shutil.copy(directory+'/tempall.txt',directory+'
            /EXECUTE_pyDAMPF/')
    shutil.copy(directory+'/tempall.txt',directory+'
            /EXECUTE_pyDAMPF/pyDAMPF_BASE/nrun/')
```

The variables inside the argument of the function *inputs_processor* are interactively requested from a shell command line. Then the file *tempall.txt* is generated and copied to the folders that will contain the simulations.

Execute pyDAMPF

For execution in a single or multi-thread way, we require first to wrap our numeric core from Fortran to Python by using f2py [Vea20]. Namely, the file *pyDAMPF.f90* within the folder *EXECUTE_pyDAMPF*.

Compilation with f2py: This step is only required once and depends on the computer architecture the code for this reads:

```
f2py -c --fcompiler=gnu95 pyDAMPF.f90 -m mypyDAMPF
```

This command-line generates mypyDAMPF.so, which will be automatically located in the simulation folders.

Once we have obtained the numerical code as Python modules, we need to choose the execution mode, which can be serial or parallel. Whereby parallel refers to multi-threading capabilities only within this first version of the code.

Serial method: This method is completely transparent to the user and will execute all the simulation cases found in the file *tempall.txt* by running the script *inputs_processor.py*. Our in-house development creates an individual folder for each simulation case, which can be executed in one thread.

```
def serial_method(tcases, factor, tempall):
    lst = gen_limites(tcases, factor)
    change_dir()
    for i in range(1,factor+1):
            direc = os.getcwd()
            direc2 = direc+'/pyDAMPF_BASE/'
            direc3 = direc+'/SERIALBASIC_0/'+str(i)+'/'
            shutil.copytree ( direc2,direc3)
    os.chdir ( direc+'/SERIALBASIC_0/1/nrun/')
    exec(open('generate_cases.py').read())
```

As arguments, the serial method requires the total number of simulation cases obtained from *tempall.txt*. In contrast, the *factor* parameter has, in this case,a default value of 1.

Parallel method: The parallel method uses more than one computational thread. It is similar to the serial method; however, this method distributes the total load along the available threads and executes in a parallel-fashion. This method comprises two parts: first, a function that takes care of the bookkeeping of cases and folders:

```
def Parallel_method(tcases, factor, tempall):
    lst = gen_limites(tcases, factor)
    change_dir()
    for i in range(1,factor+1):
        lim_inferior=lst[i-1][0]
        lim_superior=lst[i-1][1]
        direc =os.getcwd()
        direc2 =direc+'/pyDAMPF_BASE/'
        direc3 =direc+'/SERIALBASIC_0/'+str(i)+'/'
        shutil.copytree ( direc2,direc3)
        factorantiguo = ' factor=1'
        factornuevo='factor='+str(factor)
        rangoantiguo = '( 0,paraleliz)'
        rangonuevo='('+str(lim_inferior)+','
                   +str(lim_superior)+')'
        os.chdir(direc+'/PARALLELBASIC_0/'+str(i))
        pyname =' nrun/generate_cases.py'
        newpath=direc+'/PARALLELBASIC_0/'+str(i)+'/'
                   +pyname
        reemplazo(newpath,factorantiguo,factornuevo)
        reemplazo(newpath,rangoantiguo,rangonuevo)
        os.chdir(direc)
```

This part generates serial-like folders for each thread's number of cases to be executed.

The second part of the parallel method will execute pyDAMPF, which contains at the same time two scripts. One for executing pyDAMPF in a common *UNIX* based desktop or laptop. While the second is a python script that generated *SLURM* code to launch jobs in HPC facilities.

- Execution with *SLURM*

It runs pyDAMPF in different threads under the *SLURM* queuing system.

```
def cluster(factor):
    for i in range(1,factor+1):
        with open('jobpyDAMPF'+str(i)+'.x','w')
                    as ssf :
            ssf.write('#/bin/bashl|n ')
            ssf.write('#SBATCH--time=23:00:00
    \n')
            ssf.write('#SBATCH--constraint=
    epyc3\n')
```

```
            ssf.write('\n')
            ssf.write('ml Anaconda3/2019.10\n')
            ssf.write('\n')
            ssf.write('ml foss/2018a\n')
            ssf.write('\n')
            ssf.write('cd/home/$<USER>/pyDAMPF/
    EXECUTE_pyDAMPF/PARALLELBASIC_0/'+str(i)+'/nrun
    \n')
            ssf.write('\n')
            ssf.write('echo$pwd\n')
            ssf.write('\n')
            ssf.write('python3 generate_cases.py
    \n')
            ssf.close();
        os.system(sbatch jobpyDAMPF)'+str(i)+'
    .x;')
        os.system(rm jobpyDAMPF)'+str(i)+'.x;')
```

The above script generates *SLURM* jobs for a chosen set of threads; after launched, those jobs files are erased in order to improve bookkeeping.

- Parallel execution with *UNIX* based Laptops or Desktops

Usually, microscopes (AFM) computers have no SLURM pre-installed; for such a configuration, we run the following script:

```
def compute(factor):
    direc = os.getcwd()
    for i in range(1,factor+1):
        os.chdir(direc+'/PARALLELBASIC_0/'+
                 str(i)+'/nrun')
        os.system('python3 generate_cases.py
                  &')
        os.chdir(direc)
```

This function allows the proper execution of the parallel case without a queuing system and where a slight delay might appear from thread to thread execution.

Analysis

Graphically:

- With static graphics, as shown in Figures 5, 9, 13 and 17.

```
python3 Graphical_analysis.py
```

- With interactive graphics, as shown in Figure 18.

```
pip install plotly
```

```
jupyter notebook Graphical_analysis.ipynb
```

Quantitatively:

- With static data table:

```
python3 Quantitative_analysis.py
```

- With interactive tables
  Quantitative_analysis.ipynb uses a minimalistic dashboard application for tabular data visualization tabloo with easy installation.:

```
pip install tabloo
```

```
jupyter notebook Quantitative_analysis.ipynb
```

Results and discussions

In Figure 1, we show four scenarios to be tackled in this test-case for pyDAMPF. As described in the introduction, the first scenario (Figure 1), compares between air and moist environment, the second tackles soft and stiff cantilevers(see Figure 2), next is Figure Figure 3, with the cantilever volume comparison and

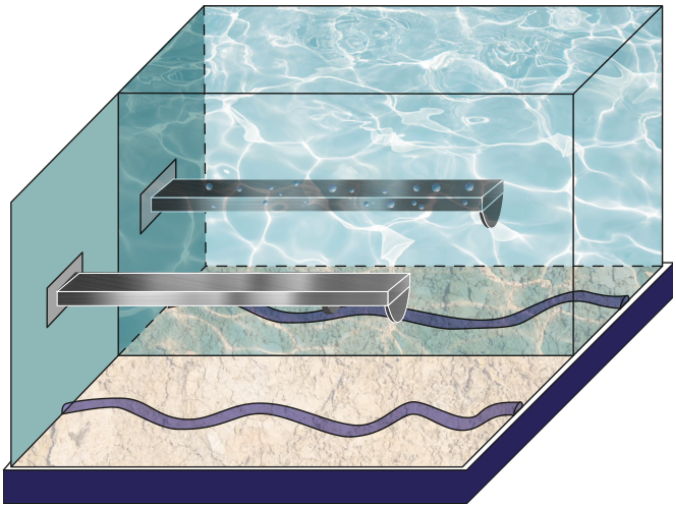


Fig. 1: *Schematic of the tip-sample interface comparing air at a given Relative Humidity with air.*

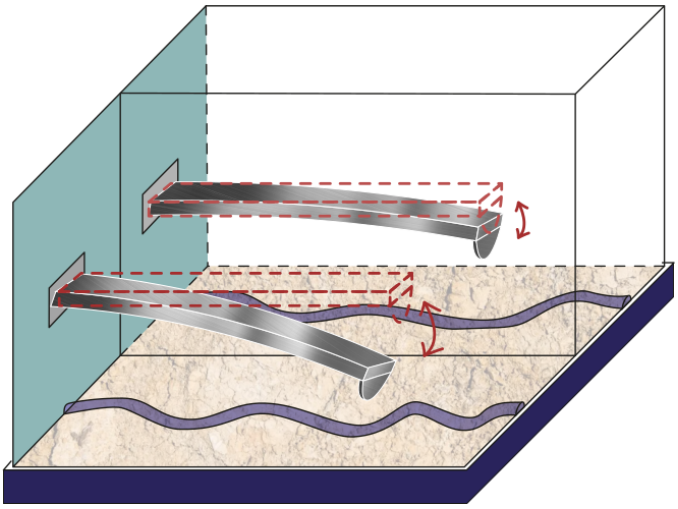


Fig. 2: *Schematic of the tip-sample interface comparing a hard (stiff) cantilever with a soft cantilever.*

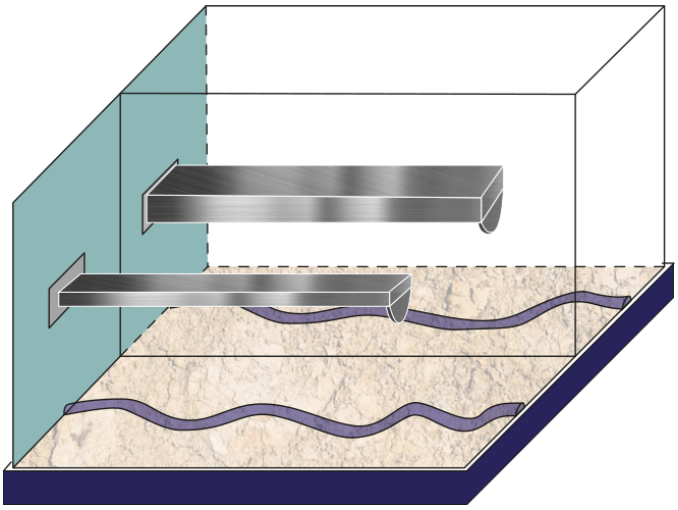


Fig. 3: *Schematic of the tip-sample interface comparing a cantilever with a high volume compared with a cantilever with a small volume.*

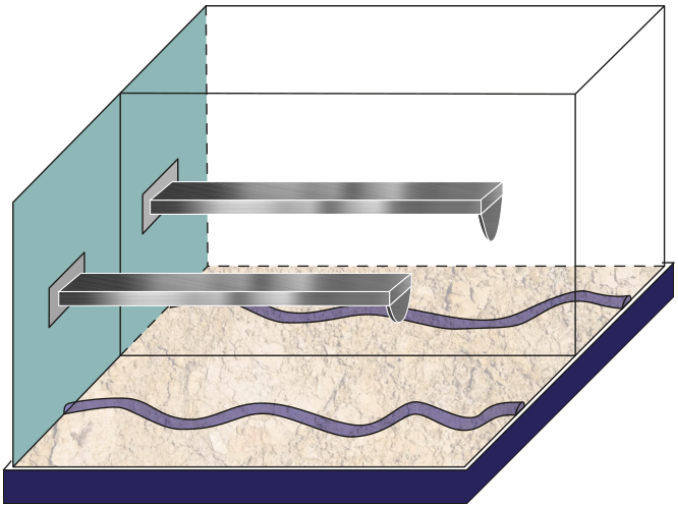


Fig. 4: *Schematic of the tip-sample interface comparing a cantilever with a wide tip with a cantilever with a sharp tip.*

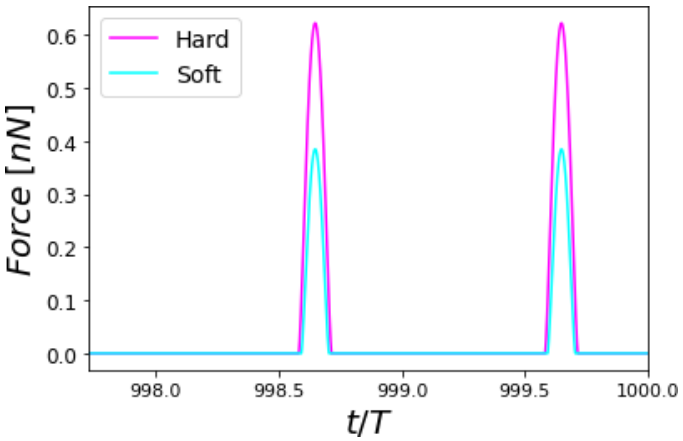


Fig. 6: *Time-varying force for PVA at RH = 60.1% for different cantilevers. The simulations show elastic (Hertz) responses. For each curve, the maximum force value is the peak force. Two complete oscillations are shown corresponding to a hard (stiff) cantilever with a soft cantilever. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

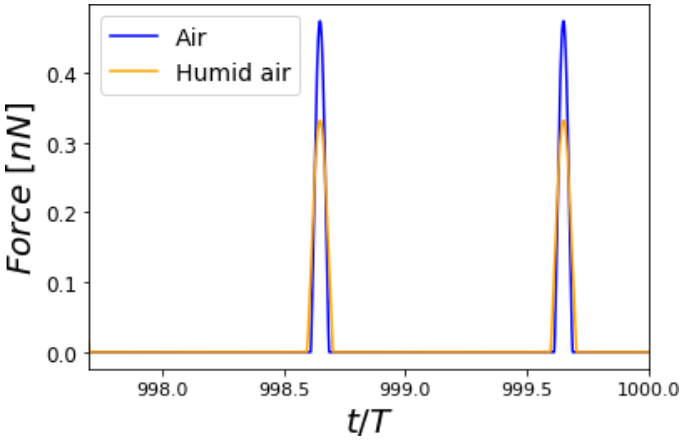


Fig. 5: *Time-varying force for PVA at RH = 60.1% for different cantilevers. The simulations show elastic (Hertz) responses. For each curve, the maximum force value is the peak force. Two complete oscillations are shown corresponding to air at a given Relative Humidity with air. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

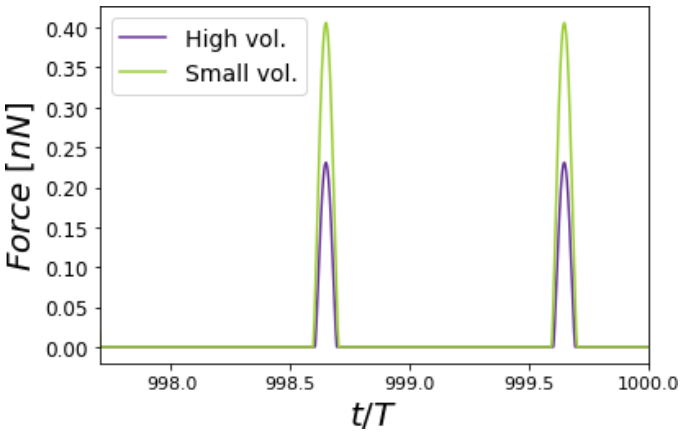


Fig. 7: *Time-varying force for PVA at RH = 60.1% for different cantilevers. The simulations show elastic (Hertz) responses. For each curve, the maximum force value is the peak force. Two complete oscillations are shown corresponding to a cantilever with a high volume compared with a cantilever with a small volume. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

the force the tip's radio (see Figure 4). Further details of the cantilevers depicted here are included in Table 22.

The AFM is widely used for mechanical properties mapping of matter [Gar20]. Hence, the first comparison of the four scenarios points out to the force response versus time according to a Hertzian interaction [Guz17]. In Figure 5, we see the humid air (RH = 60.1%) changes the measurement conditions by almost 10%. Using a stiffer cantilever ($k_c = 2.7[N/m]$) will also increase the force by almost 50% from the softer one ($k_c = 0.8[N/m]$), see Figure 6. Interestingly, the cantilever's volume, a smaller cantilever, results in the highest force by almost doubling the force by almost five folds of the smallest volume (Figure 7). Finally, the Tip radius difference between 8 and 20 nm will impact the force in roughly 40 pN (Figure 8).

Now, if we consider literature values for different RH [FCK+12], [HLLB09], we can evaluate the Peak or Maximum Forces. This force in all cases depicted in Figure 9 shows a monotonically increasing behavior with the higher Young modulus. Remarkably, the force varies in a range of 25% from dried

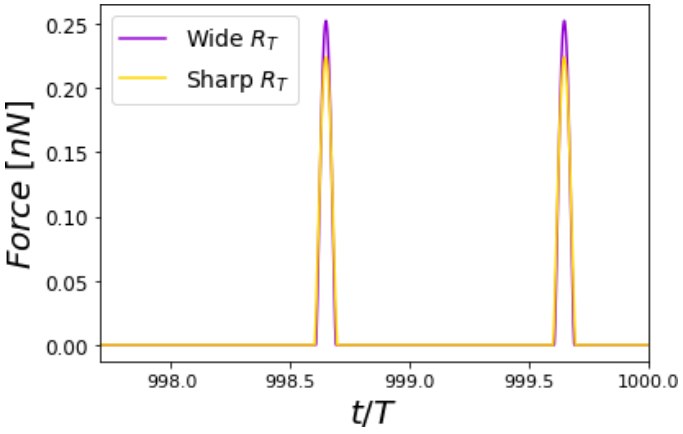


Fig. 8: *Time-varying force for PVA at RH = 60.1% for different cantilevers. The simulations show elastic (Hertz) responses. For each curve, the maximum force value is the peak force. Two complete oscillations are shown corresponding to a cantilever with a wide tip with a cantilever with a sharp tip. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

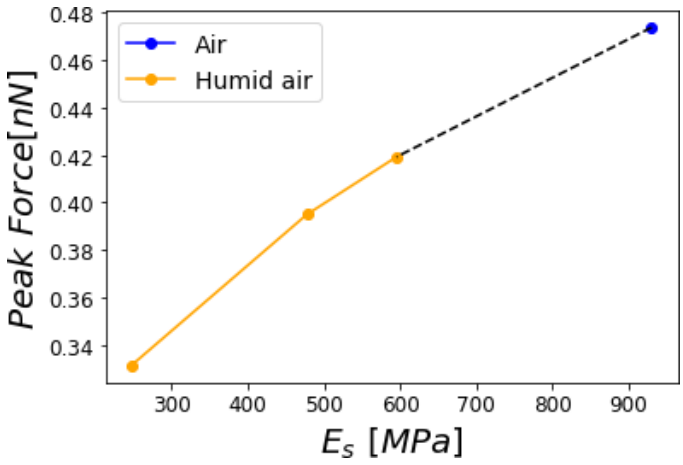


Fig. 9: *Peak force reached for a PVA sample subjected to different relative humidities 0.0%, 29.5%, 39.9% and 60.1% corresponding to air at a given Relative Humidity with air. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

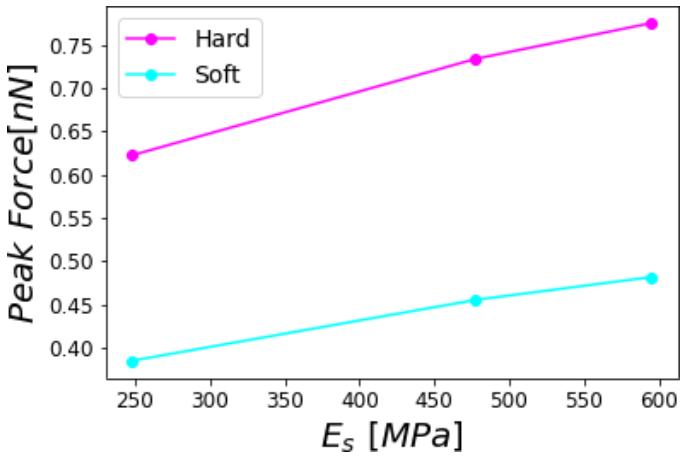


Fig. 10: *Peak force reached for a PVA sample subjected to different relative humidities 0.0%, 29.5%, 39.9% and 60.1% corresponding to a hard (stiff) cantilever with a soft cantilever. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

PVA to one at RH = 60.1% (see Figure 9).

In order to properly describe operational parameters in dynamic AFM we analyze the peak force dependence with the set-point amplitude A_{sp}. In Figure 13, we have the comparison of peak forces for the different cantilevers as a function of A_{sp}. The sensitivity of the peak force is higher for the type of cantilevers with varying k_c and V_c. Nonetheless, the peak force dependence given by the Hertzian mechanics has a dependence with the square root of the tip radius, and for those Radii on Table 22 are not influencing the force much. However, they could strongly influence resolution [GG13].

Figure 17 shows the dependence of the peak force as a function of k_c, V_c, and R_T, respectively, for all the cantilevers listed in Table 22; constituting a graphical summary of the seven analyzed cantilevers for completeness of the analysis.

Another way to summarize the results in AFM simulations if to show the Force vs. Distance curves (see Fig. 18), which in these case show exactly how for example a stiffer cantilever may penetrate more into the sample by simple checking the distance cantilever e reaches. On the other hand, it also jumps into the

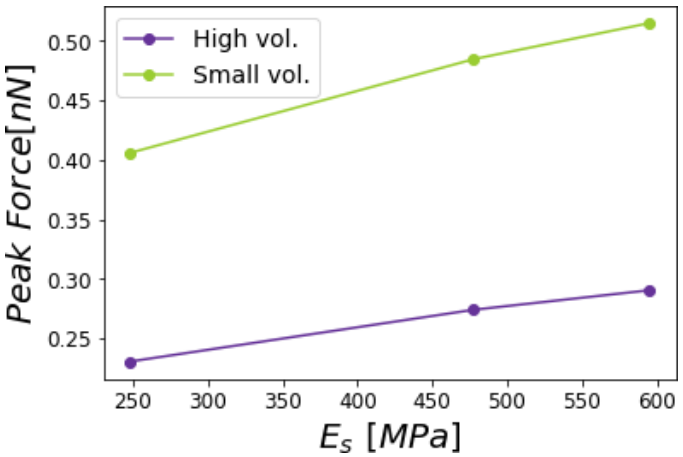


Fig. 11: *Peak force reached for a PVA sample subjected to different relative humidities 0.0%, 29.5%, 39.9% and 60.1% corresponding to a cantilever with a high volume compared with a cantilever with a small volume. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

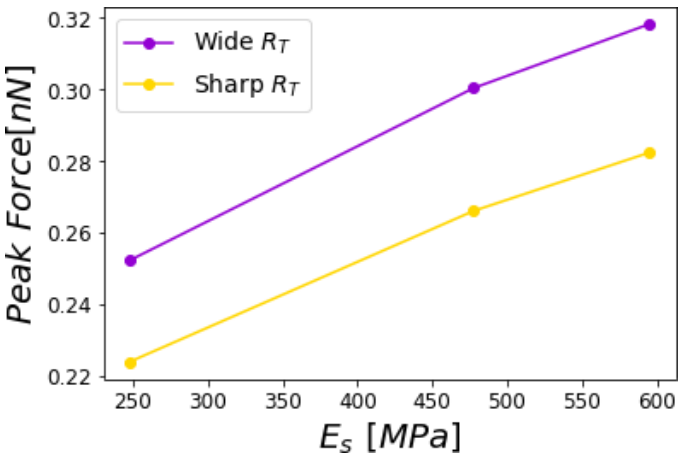


Fig. 12: *Peak force reached for a PVA sample subjected to different relative humidities 0.0%, 29.5%, 39.9% and 60.1% corresponding to a cantilever with a wide tip with a cantilever with a sharp tip. The simulations were performed for $A_{sp}/A_0 = 0.8$.*

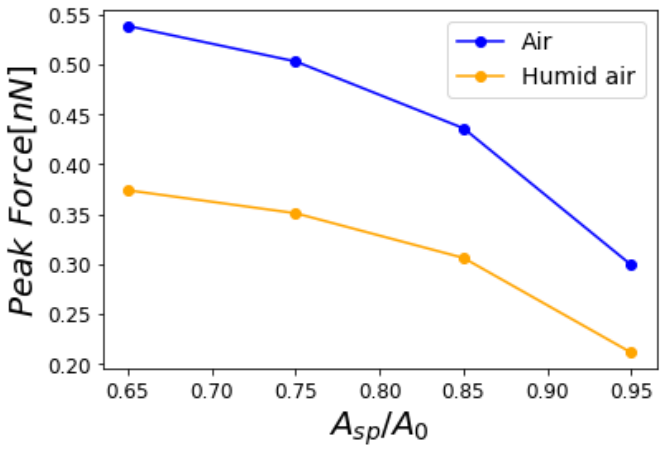


Fig. 13: *Dependence of the maximum force on the set-point amplitude corresponding to air at a given Relative Humidity with air.*

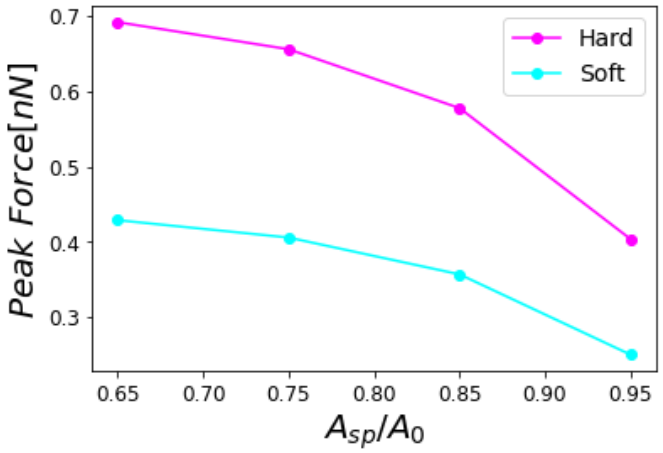


Fig. 14: *Dependence of the maximum force on the set-point amplitude corresponding to a hard (stiff) cantilever with a soft cantilever.*

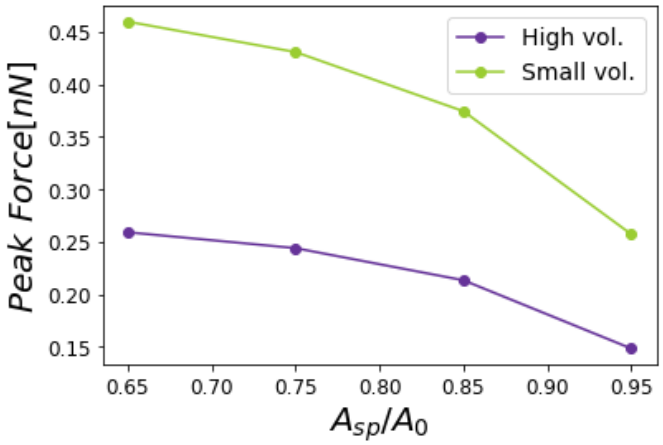


Fig. 15: *Dependence of the maximum force on the set-point amplitude corresponding to a cantilever with a high volume compared with a cantilever with a small volume.*

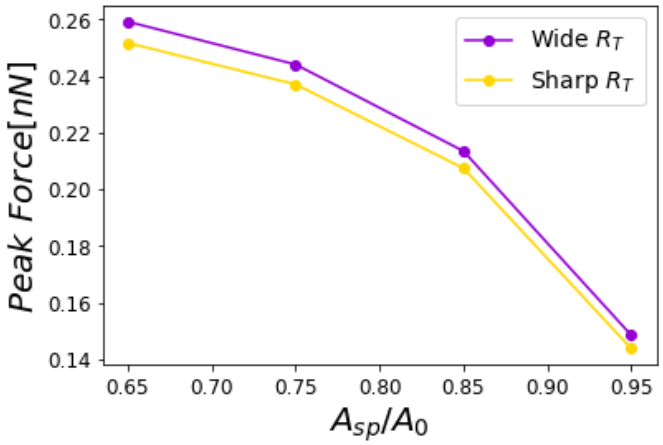


Fig. 16: *Dependence of the maximum force on the set-point amplitude corresponding to a cantilever with a wide tip with a cantilever with a sharp tip.*

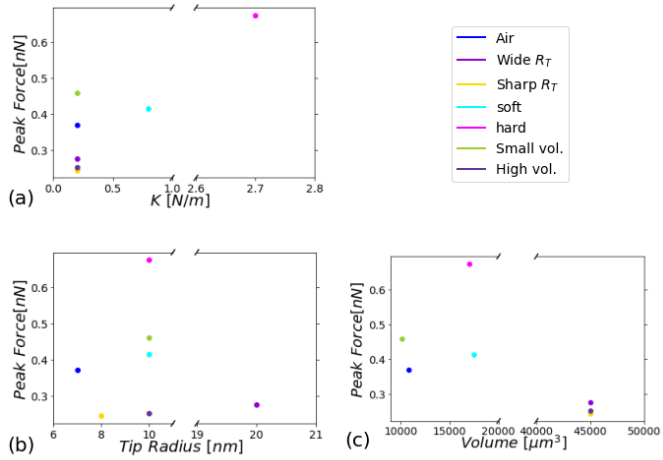


Fig. 17: *Dependence of the maximum force with the most important characteristics of each cantilever, filtering the cantilevers used for the scenarios , the figure shows maximum force dependent on the: (a) force constant k, (b) cantilever tip radius, and (c) cantilever volume, respectively. The simulations were performed for A_{sp}/A_{0} = 0.8.*

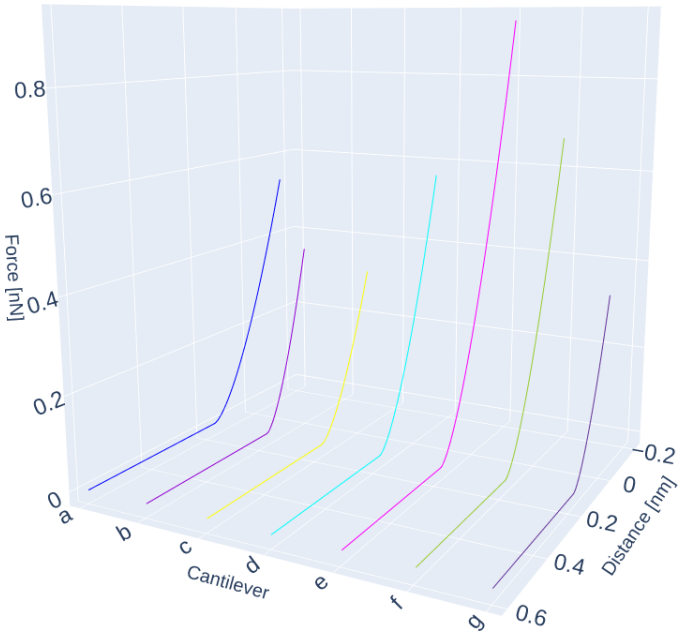


Fig. 18: *Three-dimensional plots of the various cantilevers provided by the manufacturer and those in the pyDAMPF database that establish a given maximum force at a given distance between the tip and the sample for a PVA polymer subjected to RH= 0% with E = 930 [MPa].*

eyes that a cantilever with small volume *f* has less damping from the environment and thus it also indents more than the ones with higher volume. Although these type of plots are the easiest to make, they carry lots of experimental information. In addition, pyDAMPF can plot such 3D figures interactively that enables a detailed comparison of those curves.

As we aim a massive use of pyDAMPF, we also perform the corresponding benchmarks on four different computing platforms, where two of them resembles the standard PC or Laptop found at the labs, and the other two aim to cloud and HPC facilities,

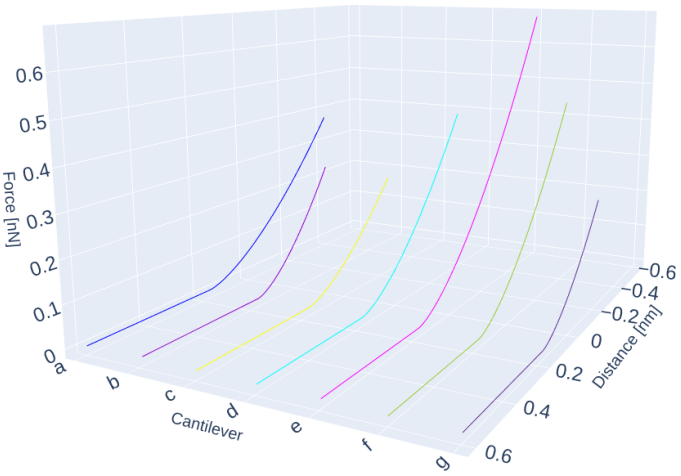


Fig. 19: *Three-dimensional plots of the various cantilevers provided by the manufacturer and those in the pyDAMPF database that establish a given maximum force at a given distance between the tip and the sample for a PVA polymer subjected to RH = 60.1% with E = 248.8 [MPa].*

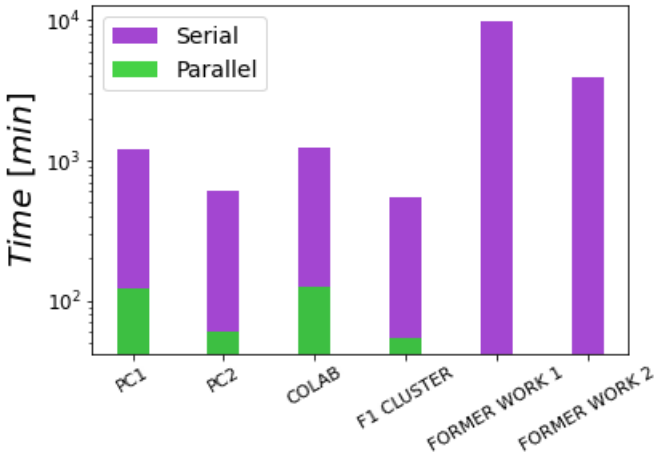


Fig. 20: *Comparison of times taken by both the parallel method and the serial method.*

respectively (see Table 23 for details).

Figure 20 shows the average run time for the serial and parallel implementation. Despite a slightly higher performance for the case of the HPC cluster nodes, a high-end computer (PC 2) may also reach similar values, which is our current goal. Another striking aspect observed by looking at the speed-up, is the maximum and minimum run times, which notoriously show the on-demand character of cloud services. As their maxima and minima show the highest variations.

To calculate the speed up we use the following equation:

$$S = \frac{t_{total}}{t_{thread}}$$

Where S is the speed up , t_{Thread} is the execution time of a computational thread, and t_{Total} is the sum of times, shown in the table 24. For our calculations we used the highest, the average and the lowest execution time per thread.

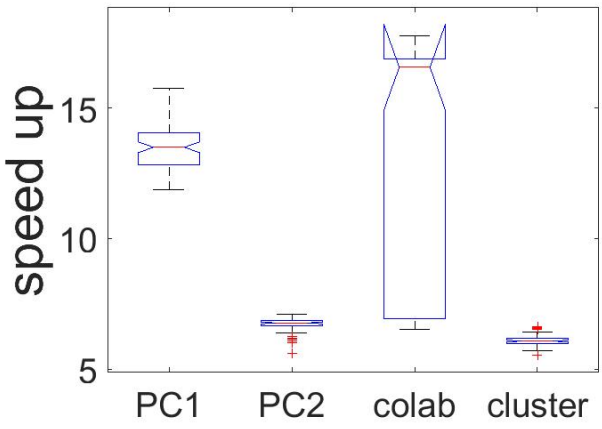


Fig. 21: *Speed up parallel method.*

Scenario	Model	R[nm]	Q^{**}	E_s[MPa]	k[N/m]	f[kHz]	Volume [μm^3]	Cantilever
Air	PPP-CONTSCR	7	28.65	930	0.2	25	10800	a
Humid air	PPP-CONTSCR	7	28.74 - 28.77 - 28.83	595.6 - 477.8 - 247.8	0.2	25	10800	a
Hard	XCS11-B	10	167.73 - 167.89 - 168.28	595.6 - 477.8 - 247.8	2.7	80	17010	e
Soft	HQ:CSC37-A	10	95.15 - 95.24 - 94.45	595.6 - 477.8 - 247.8	0.8	40	17500	d
High vol.	PPP-XY-CONTR	10	48.37 - 48.41 - 48.51	595.6 - 477.8 - 247.8	0.2	13	45000	g
Small vol.	SD-PXL-CONT-SC	10	23.61 - 23.64 - 23.68	595.6 - 477.8 - 247.8	0.2	8	10125	f
Wide R_T	SD-CONT-SiN*	20	48.37 - 48.41 - 48.51	595.6 - 477.8 - 247.8	0.2	13	45000	b
Sharp R_T	CONTR	8	48.37 - 48.41 - 48.51	595.6 - 477.8 - 247.8	0.2	13	45000	c

Fig. 22: *Data used for Figs. 5, 9 and 13 with an $A_0 = 10[nm]$. Observe that the quality factor and Young's modulus have three different values respectively for RH1 = 29.5%, RH2 = 39.9% y RH3 = 60.1%. ** The values presented for Quality Factor Q were calculated at Google Colaboratory notebook Q calculation, using the method proposed by [GS05], [Sad98].*

Computer	CPU	RAM	pyDAMPF	Forme Work
PC 1	Core i5-5200U @ 2.2 GHz	8 GB	✓	✓
PC 2	Ryzen 7 4800H @ 2.9 GHz	16GB	✓	✓
Colab	Intel(R) Xeon(R) CPU @ 2.2GHz	6.4GB*	✓	-
Cluster	Intel(R) Xeon(R) E5-2620 v4 @ 2.1GHz	32 GB	✓	-

Fig. 23: *Computers used to run pyDAMPF and Former work [GGG15], * the free version of Colab provides this capability, there are two paid versions which provide much greater capacity, these versions known as Colab Pro and Colab Pro+ are only available in some countries.*

Time of execution for thread [min]				
Thread	PC 1	PC 2	Colab	Cluster
1	123.81	60.19	151.10	55.19
2	116.12	61.23	151.94	54.96
3	123.73	60.09	148.94	52.87
4	120.41	60.26	62.01	54.24
5	122.27	60.35	61.39	55.48
6	119.93	59.97	59.91	52.96
7	122.70	60.68	149.89	54.72
8	124.19	60.55	166.58	54.54
9	121.34	60.99	136.69	54.36
10	123.36	60.57	152.88	57.75
Average	121.78	60.49	124.13	54.71
Sum	1217.85	604.89	1241.33	547.07

Fig. 24: *Execution times per computational thread, for each computer. Note that each Thread consists of 9 simulation cases, with a sum time showing the total of 90 cases for evaluating 3 different Young moduli and 30 cantilevers at the same time.*

Limitations

The main limitation of dynamic AFM simulators based in continuum modeling is that sometimes a molecular behavior is overlooked. Such a limitation comes from the multiple time and length scales behind the physics of complex systems, as it is the case of polymers and biopolymers. In this regard, several efforts on the multiscale modeling of materials have been proposed, joining mainly efforts to stretch the multiscale gap [GTK+19]. We also plan to do so, within a current project, for modeling the polymeric fibers as molecular chains and providing "feedback" between models from a top-down strategy. Code-wise, the implementation will be also gradually improved. Nonetheless, to maintain scientific code is a challenging task. In particular without the support for our students once they finish their thesis. In this respect, we will seek software funding and more community contributions.

Future work

There are several improvements that are planned for pyDAMPF.

- We plan to include a link to molecular dynamics simulations of polymer chains in a multiscale like approach.
- We plan to use experimental values with less uncertainty to boost semi-empirical models based on pyDAMPF.
- The code is still not very clean and some internal cleanup is necessary. This is especially true for the Python backend which may require a refactoring.
- Some AI optimization was also envisioned, particularly for optimizing criteria and comparing operational parameters.

Conclusions

In summary, pyDAMPF is a highly efficient and adaptable simulation tool aimed at analyzing, planning and interpreting dynamic AFM experiments.

It is important to keep in mind that pyDAMPF uses cantilever manufacturers information to analyze, evaluate and choose a certain nanoprobe that fulfills experimental criteria. If this will not be the case, it will advise the experimentalists on what to expect from their measurements and the response a material may have. We currently support multi-thread execution using in-house development. However, in our outlook, we plan to extend the code to GPU by using transpiling tools, like compyle [Ram20], as the availability of GPUs also increases in standard workstations. In addition, we have shown how to reuse a widely tested Fortran code [GPG13] and wrap it as a python module to profit from pythonic libraries and interactivity via Jupyter notebooks. Implementing new interaction forces for the simulator is straightforward. However, this code includes the state-of-the-art contact, viscous, van der Waals, capillarity and electrostatic forces used for physics at the interfaces. Moreover, we plan to implement soon semi-empirical analysis and multiscale modeling with molecular dynamics simulations.

Acknowledgments

H.V.G thanks the financial support by the Slovenian Research Agency (Funding No. P1-0055). We gratefully acknowledge the fruitful discussions with Tomas Corrales and our joint Fondecyt Regular project 1211901.

REFERENCES

[FCK+12] Kathrin Friedemann, Tomas Corrales, Michael Kappl, Katharina Landfester, and Daniel Crespy. Facile and large-scale fabrication of anisometric particles from fibers synthesized by colloid electrospinning. *Small*, 8:144–153, 2012. doi:10.1002/smll.201101247.

[Gar20] Ricardo Garcia. Nanomechanical mapping of soft materials with the atomic force microscope: methods, theory and applications. *The Royal Society of Chemistry*, 49:5850–5884, 2020. doi:10.1039/d0cs00318b.

[GG13] Horacio V. Guzman and Ricardo Garcia. Peak forces and lateral resolution in amplitude modulation force microscopy in liquid. *Beilstein Journal of Nanotechnology*, 4:852–859, 2013. doi:10.3762/bjnano.4.96.

[GGG15] Horacio V. Guzman, Pablo D. Garcia, and Ricardo Garcia. Dynamic force microscopy simulator (dforce): A tool for planning and understanding tapping and bimodal afm experiments. *Beilstein Journal of Nanotechnology*, 6:369–379, 2015. doi:10.3762/bjnano.6.36.

[GPG13] Horacio V. Guzman, Alma P. Perrino, and Ricardo Garcia. Peak forces in high-resolution imaging of soft matter in liquid. *ACS Nano*, 7:3198–3204, 2013. doi:10.1021/nn4012835.

[GS05] Christopher P. Green and John E. Sader. Frequency response of cantilever beams immersed in viscous fluids near a solid surface with applications to the atomic force microscope. *Journal of Applied Physics*, 98:114913, 2005. doi:10.1063/1.2136418.

[GTK+19] Horacio V. Guzman, Nikita Tretyakov, Hideki Kobayashi, Aoife C. Fogarty, Karsten Kreis, Jakub Krajniak, Christoph Junghans, Kurt Kremer, and Torsten Stuehn. Espresso++ 2.0: Advanced methods for multiscale molecular simulation. *Computer Physics Communications*, 238:66–76, 2019. doi:10.1016/j.cpc.2018.12.017.

[Guz17] Horacio V. Guzman. Scaling law to determine peak forces in tapping-mode afm experiments on finite elastic soft matter systems. *Beilstein Journal of Nanotechnology*, 8:968–974, 2017. doi:10.3762/bjnano.8.98.

[HLLB09] Fei Hang, Dun Lu, Shuang Wu Li, and Asa H. Barber. Stress-strain behavior of individual electrospun polymer fibers using combination afm and sem. *Materials Research Society*, 1185:1185–II07–10, 2009. doi:10.1557/PROC-1185-II07-10.

[MHR08] John Melcher, Shuiqing Hu, and Arvind Raman. Veda: A web-based virtual environment for dynamic atomic force microscopy. *Review of Scientific Instruments*, 79:061301, 2008. doi:10.1063/1.2938864.

[Ram20] Prabhu Ramachandran. Compyle: a Python package for parallel computing. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 32 – 39, 2020. doi:10.25080/majora-342d178e-005.

[Sad98] John E. Sader. Frequency response of cantilever beams immersed in viscous fluids with applications to the atomic force microscope. *Journal of Applied Physics*, 84:64–76, 1998. doi:10.1063/1.368002.

[Vea20] Pauli Virtanen and et al. Scipy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

Improving PyDDA's atmospheric wind retrievals using automatic differentiation and Augmented Lagrangian methods

Robert Jackson[‡*], Rebecca Gjini[§], Sri Hari Krishna Narayanan[‡], Matt Menickelly, Paul Hovland[‡], Jan Hückelheim[‡], Scott Collis[‡]

Introduction

Meteorologists require information about the spatiotemporal distribution of winds in thunderstorms in order to analyze how physical and dynamical processes govern thunderstorm evolution. Knowledge of such processes is vital for predicting severe and hazardous weather events. However, acquiring wind observations in thunderstorms is a non-trivial task. There are a variety of instruments that can measure winds including radars, anemometers, and vertically pointing wind profilers. The difficulty in acquiring a three dimensional volume of the 3D wind field from these sensors is that these sensors typically only measure either point observations or only the component of the wind field parallel to the direction of the antenna. Therefore, in order to obtain 3D wind fields, the weather radar community uses a weak variational technique that finds a 3D wind field that minimizes a cost function J.

$$J(\mathbf{V}) = \mu_m J_m + \mu_o J_o + \mu_v J_v + \mu_b J_b + \mu_s J_s \tag{1}$$

Here, J_m is how much the wind field $\mathbf{V}$ violates the anelastic mass continuity equation. J_o is how much the wind field is different from the radar observations. J_v is how much the wind field violates the vertical vorticity equation. J_b is how much the wind field differs from a prescribed background. Finally J_s is related to the smoothness of the wind field, quantified as the Laplacian of the wind field. The scalars μ_x are weights determining the relative contribution of each cost function to the total J. The flexibility in this formulation potentially allows for factoring in the uncertainties that are inherent in the measurements. This formulation is expandable to include cost functions related to data from other sources such as weather forecast models and soundings. For more specific information on these cost functions, see [SPG09] and [PSX12].

PyDDA is an open source Python package that implements the weak variational technique for retrieving winds. It was originally developed in order to modernize existing codes for the weak variational retrievals such as CEDRIC [MF98] and Multidop [LSKJ17] as detailed in the 2019 SciPy Conference proceedings (see [JCL+20], [RJSCTL+19]). It provided a much easier to use and more portable interface for wind retrievals than was provided by these packages. In PyDDA versions 0.5 and prior, the implementation of Equation (1) uses NumPy [HMvdW+20] to calculate J and its gradient. In order to find the wind field $\mathbf{V}$ that minimizes J, PyDDA used the limited memory Broyden–Fletcher–Goldfarb–Shanno bounded (L-BFGS-B) from SciPy [VGO+20]. L-BFGS-B requires gradients of J in order to minimize J. Considering the antiquity of the CEDRIC and Multidop packages, these first steps provided the transition to Python that was needed in order to enhance accessibility of wind retrieval software by the scientific community. For more information about PyDDA versions 0.5 and prior, consult [RJSCTL+19] and [JCL+20].

However, there are further improvements that still needed to be made in order to optimize both the accuracy and speed of the PyDDA retrievals. For example, the cost functions and gradients in PyDDA 0.5 are implemented in NumPy which does not take advantage of GPU architectures for potential speedups [HMvdW+20]. In addition, the gradients of the cost function that are required for the weak variational technique are hand-coded even though packages such as Jax [BFH+18] and TensorFlow [AAB+15] can automatically calculate these gradients. These needs motivated new features for the release of PyDDA 1.0. In PyDDA 1.0, we utilize Jax and TensorFlow's automatic differentiation capabilities for differentiating J, making these calculations less prone to human error and more efficient.

Finally, upgrading PyDDA to use Jax and TensorFlow allows it to take advantage of GPUs, increasing the speed of retrievals. This paper shows how Jax and TensorFlow are used to automatically calculate the gradient of J and improve the performance of PyDDA's wind retrievals using GPUs.

In addition, a drawback to the weak variational technique is that the technique requires user specified constants μ. This therefore creates the possibility that winds retrieved from different datasets may not be physically consistent with each other, affecting reproducibility. Therefore, for the PyDDA 1.1 release, this paper also details a new approach that uses Augmented Lagrangian solvers in order to place strong constraints on the wind field such that it satisfies a mass continuity constraint to within a specified tolerance while minimizing the rest of the cost function. This new approach also takes advantage of the automatically calculated

* Corresponding author: rjackson@anl.gov
‡ Argonne National Laboratory, 9700 Cass Ave., Argonne, IL, 60439
§ University of California at San Diego

gradients that are implemented in PyDDA 1.0. This paper will show that this new approach eliminates the need for user specified constants, ensuring the reproducibility of the results produced by PyDDA.

Weak variational technique

This section summarizes the weak variational technique that was implemented in PyDDA previous to version 1.0 and is currently the default option for PyDDA 1.1. PyDDA currently uses the weak variational formulation given by Equation (1). For this proceedings, we will focus our attention on the mass continuity J_m and observational cost function J_o. In PyDDA, J_m is given as the discrete volume integral of the square of the anelastic mass continuity equation

$$J_m(u,v,w) = \sum_{volume} \left[\frac{\delta(\rho_s u)}{\delta x} + \frac{\delta(\rho_s v)}{\delta y} + \frac{\delta(\rho_s w)}{\delta z} \right]^2, \quad (2)$$

where u is the zonal component of the wind field and v is the meridional component of the wind field. ρ_s is the density of air, which is approximated in PyDDA as $\rho_s(z) = e^{-z/10000}$ where z is the height in meters. The physical interpretation of this equation is that a column of air in the atmosphere is only allowed to compress in order to generate changes in air density in the vertical direction. Therefore, wind convergence at the surface will generate vertical air motion. A corollary of this is that divergent winds must occur in the presence of a downdraft. At the scales of winds observed by PyDDA, this is a reasonable approximation of the winds in the atmosphere.

The cost function J_o metricizes how much the wind field is different from the winds measured by each radar. Since a scanning radar will scan a storm while pointing at an elevation angle θ and an azimuth angle ϕ, the wind field must first be projected to the radar's coordinates. After that, PyDDA finds the total square error between the analysis wind field and the radar observed winds as done in Equation (3).

$$J_o(u,v,w) = \sum_{volume} (u\cos\theta\sin\phi + v\cos\theta\cos\phi + (w - w_t)\sin\theta)^2 \quad (3)$$

Here, w_t is the terminal velocity of the particles scanned by the radar volume. This is approximated using empirical relationships between w_t and the radar reflectivity Z. PyDDA then uses the limited memory Broyden–Fletcher–Goldfarb–Shanno bounded (L-BFGS-B) algorithm (see, e.g., [LN89]) to find the u, v, and w that solves the optimization problem

$$\min_{u,v,w} J(u,v,w) \triangleq \mu_m J_m(u,v,w) + \mu_v J_v(u,v,w). \quad (4)$$

For experiments using the weak variational technique, we run the optimization until either the $L^{\inf}$ norm of the gradient of J is less than 10^{-8} or when the maximum change in u, v, and w between iterations is less than 0.01 m/s as done by [PSX12]. Typically, the second criteria is reached first. Before PyDDA 1.0, PyDDA utilized SciPy's L-BFGS-B implementation. However, as of PyDDA 1.0 one can also use TensorFlow's L-BFGS-B implementation, which is used here for the experiments with the weak variational technique [AAB+15].

Using automatic differentiation

The optimization problem in Equation (4) requires the gradients of J. In PyDDA 0.5 and prior, the gradients of the cost function J were calculated by finding the closed form of the gradient by hand and then coding the closed form in Python. The code snippet below provides an example of how the cost function J_m is implemented in PyDDA using NumPy.

```
def calculate_mass_continuity(u, v, w, z, dx, dy, dz):

    dudx = np.gradient(u, dx, axis=2)
    dvdy = np.gradient(v, dy, axis=1)
    dwdz = np.gradient(w, dz, axis=0)

    div = dudx + dvdy + dwdz

    return coeff * np.sum(np.square(div)) / 2.0
```

In order to hand code the gradient of the cost function above, one has to write the closed form of the derivative into another function like below.

```
def calculate_mass_continuity_gradient(u, v, w, z, dx,
                                       dy, dz, coeff):
    dudx = np.gradient(u, dx, axis=2)
    dvdy = np.gradient(v, dy, axis=1)
    dwdz = np.gradient(w, dz, axis=0)

    grad_u = -np.gradient(div, dx, axis=2) * coeff
    grad_v = -np.gradient(div, dy, axis=1) * coeff
    grad_w = -np.gradient(div, dz, axis=0) * coeff

    y = np.stack([grad_u, grad_v, grad_w], axis=0)
    return y.flatten()
```

Hand coding these functions can be labor intensive for complicated cost functions. In addition, there is no guarantee that there is a closed form solution for the gradient. Therefore, we tested using both Jax and TensorFlow to automatically compute the gradients of J. Computing the gradients of J using Jax can be done in two lines of code using `jax.vjp`:

```
primals, fun_vjp = jax.vjp(
    calculate_radial_vel_cost_function,
    vrs, azs, els, u, v, w, wts, rmsVr, weights,
    coeff)
_, _, _, p_x1, p_y1, p_z1, _, _, _, _ = fun_vjp(1.0)
```

Calculating the gradients using automatic differentiation using TensorFlow is also a simple code snippet using `tf.GradientTape`:

```
with tf.GradientTape() as tape:
    tape.watch(u)
    tape.watch(v)
    tape.watch(w)
    loss = calculate_radial_vel_cost_function(
        vrs, azs, els, u, v, w,
        wts, rmsVr, weights, coeff)

grad = tape.gradient(loss)
```

As one can see, there is no more need to derive the closed form of the gradient of the cost function. Rather, the cost function itself is now the input to a snippet of code that automatically provides the derivative. In PyDDA 1.0, there are now three different engines that the user can specify. The classic "scipy" mode uses the NumPy-based cost function and hand coded gradients used by versions of PyDDA previous to 1.0. In addition, there are now TensorFlow and Jax modes that use both cost functions and automatically generated gradients generated using TensorFlow or Jax.

Improving performance with GPU capabilities

The implementation of a TensorFlow-based engine provides PyDDA the capability to take advantage of CUDA-compatible GPUs.

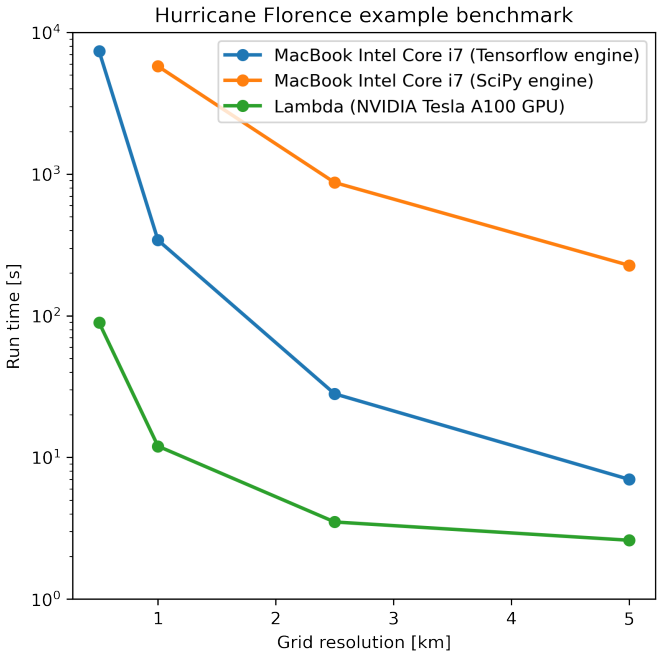


Fig. 1: *The time in seconds of execution of the Hurricane Florence retrieval example when using the TensorFlow and SciPy engines on an Intel Core i7 MacBook in CPU mode and on a node of Argonne National Laboratory's Lambda cluster, utilizing a single NVIDIA Tesla A100 GPU for the calculation.*

Method	***0.5 km***	***1 km***	***2.5 km***	***5.0 km***
SciPy Engine	~50 days	5771.2 s	871.5 s	226.9 s
TensorFlow Engine	7372.5 s	341.5 s	28.1 s	7.0 s
NVIDIA Tesla A100 GPU	89.4 s	12.0 s	3.5 s	2.6 s

TABLE 1: *Run times for each of the benchmarks in Figure 1.*

Given that weather radar datasets can span decades and processing each 10 minute time period of data given by the radar can take on the order of 1-2 minutes with PyDDA using regular CPU operations, if this time were reduced to seconds, then processing winds from years of radar data would become tenable. Therefore, we used the TensorFlow-based PyDDA using the weak variational technique on the Hurricane Florence example in the PyDDA Documentation. On 14 September 2018, Hurricane Florence was within range of 2 radars from the NEXRAD network: KMHX stationed in Newport, NC and KLTX stationed in Wilmington, NC. In addition, the High Resolution Rapid Refresh model runs provided an additional constraint for the wind retrieval. For more information on this example, see [RJSCTL+19]. The analysis domain spans 400 km by 400 km horizontally, and the horizontal resolution was allowed to vary for different runs in order to compare how both the CPU and GPU-based retrievals' performance would be affected by grid resolution. The time of completion of each of these retrievals is shown in Figure 1.

Figure 1 and Table 1 show that, in general, the retrievals took anywhere from 10 to 100 fold less time on the GPU compared to the CPU. The discrepancy in performance between the GPU and CPU-based retrievals increases as resolution decreases, demonstrating the importance of the GPU for conducting high-resolution wind retrievals. In Table 1, using a GPU to retrieve the Hurricane Florence example at 1 km resolution reduces the run time from 341 s to 12 s. Therefore, these performance improvements show that PyDDA's TensorFlow-based engine now enables it to handle both spatial scales of hundreds of kms at a 1 km resolution. For a day of data at this resolution, assuming five minutes between scans, an entire day of data can be processed in 57 minutes. With the use of multi-GPU clusters and selecting for cases where precipitation is present, this enables the ability to process winds from multi-year radar datasets within days instead of months.

In addition, simply using TensorFlow's implementation of L-BFGS-B as well as the TensorFlow calculated cost function and gradients provides a significant performance improvement compared to the original "scipy" engine in PyDDA 0.5, being up to a factor of 30 faster. In fact, running PyDDA's original "scipy" engine on the 0.5 km resolution data for the Hurricane Florence example would have likely taken 50 days to complete on an Intel Core i7-based MacBook laptop. Therefore, that particular run was not tenable to do and therefore not shown in Figure 1. In any case, this shows that upgrading the calculations to use TensorFlow's automatically generated gradients and L-BFGS-B implementation provides a very significant speedup to the processing time.

Augmented Lagrangian method

The release of PyDDA 1.0 focused on improving its performance and gradient accuracy by using automatic differentiation for calculating the gradient. For PyDDA 1.1, the PyDDA development team focused on implementing a technique that enables the user to automatically determine the weight coefficients μ. This technique builds upon the automatic differentiation work done for PyDDA 1.0 by using the automatically generated gradients. In this work, we consider a constrained reformulation of Equation (4) that requires wind fields returned by PyDDA to (approximately) satisfy mass continuity constraints. That is, we focus on the constrained optimization problem

$$\begin{aligned} \min_{u,v,w} \quad & J_v(u,v,w) \\ \text{s. to} \quad & J_m(u,v,w) = 0, \end{aligned} \tag{5}$$

where we now interpret J_m as a vector mapping that outputs, at each grid point in the discretized volume $\frac{\delta(\rho_s u)}{\delta x} + \frac{\delta(\rho_s v)}{\delta y} + \frac{\delta(\rho_s w)}{\delta z}$. Notice that the formulation in Equation (5) has no dependencies on scalars μ.

To solve the optimization problem in Equation (5), we implemented an augmented Lagrangian method with a filter mechanism inspired by [LV20]. An augmented Lagrangian method considers the Lagrangian associated with an equality-constrained optimization problem, in this case $\mathscr{L}_0(u,v,w,\lambda) = J_v(u,v,w) - \lambda^\top J_m(u,v,w)$, where λ is a vector of Lagrange multipliers of the same length as the number of grid points in the discretized volume. The Lagrangian is then *augmented* with an additional squared-penalty term on the constraints to yield $\mathscr{L}_\mu(u,v,w,\lambda) = \mathscr{L}_0(u,v,w,\lambda) + \frac{\mu}{2}\|J_m(u,v,w)\|^2$, where we have intentionally used $\mu > 0$ as the scalar in the penalty term to make comparisons with Equation (4) transparent. It is well known (see, for instance, Theorem 17.5 of [NW06]) that under some not overly restrictive conditions there exists a finite $\bar{\mu}$ such that if $\mu \geq \bar{\mu}$, then each local solution of Equation (5) corresponds to a strict local minimizer

of $\mathscr{L}_\mu(u,v,w,\lambda^*)$ for a suitable choice of multipliers λ^*. Essentially, augmented Lagrangian methods solve a short sequence of unconstrained problems $\mathscr{L}_\mu(u,v,w,\lambda)$, with different values of μ until a solution is returned that is a local, feasible solution to Equation (5). In our implementation of an augmented Lagrangian method, the coarse minimization of $\mathscr{L}_\mu(u,v,w,\lambda)$ is performed by the Scipy implementation of LBFGS-B with the TensorFlow implementation of the cost function and gradients. Additionally, in our implementation, we employ a filter mechanism (see a survey in [FLT06]) recently proposed for augmented Lagrangian methods in [LV20] in order to guarantee convergence. We defer details to that paper, but note that the feasibility restoration phase (the minimization of a squared constraint violation) required by such a filter method is also performed by the SciPy implementation of LBFGS-B.

The PyDDA documentation contains an example of a mesoscale convective system (MCS) that was sampled by a C-band Polarization Radar (CPOL) and a Bureau of Meteorology Australia radar on 20 Jan 2006 in Darwin, Australia. For more details on this storm and the radar network configuration, see [CPMW13]. For more information about the CPOL radar dataset, see [JCL+18]. This example with its data is included in the PyDDA Documentation as the "Example of retrieving and plotting winds."

Figure 2 shows the winds retrieved by the Augmented Lagrangian technique with $\mu = 1$ and from the weak variational technique with $\mu = 1$ on the right. Figure 2 shows that both techniques are capturing similar horizontal wind fields in this storm. However, the Augmented Lagrangian technique is resolving an updraft that is not present in the wind field generated by the weak variational technique. Since there is horizontal wind convergence in this region, we expect there to be an updraft present in this box in order for the solution to be physically realistic. Therefore, for $\mu = 1$, the Augmented Lagrangian technique is doing a better job at resolving the updrafts present in the storm than the weak variational technique is. This shows that adjusting μ is required in order for the weak variational technique to resolve the updraft.

We solve the unconstrained formulation (4) using the implementation of L-BFGS-B currently employed in PyDDA; we fix the value $\mu_v = 1$ and vary $\mu_m = 2^j : j = 0,1,2,\ldots,16$. We also solve the constrained formulation (5) using our implementation of a filter Augmented Lagrangian method, and instead vary the initial guess of penalty parameter $\mu = 2^j : j = 0,1,2,\ldots,16$. For the initial state, we use the wind profile from the weather balloon launch at 00 UTC 20 Jan 2006 from Darwin and apply it to the whole analysis domain. A summary of results is shown in Figures 3 and 4. We applied a maximum constraint violation tolerance of 10^{-3} to the filter Augmented Lagrangian method. This is a tolerance that assumes that the winds do not violate the mass continuity constraint by more than 0.001 $m^2 s^{-2}$. Notice that such a tolerance is impossible to supply to the weak variational method, highlighting the key advantage of employing a constrained method. Notice that in this example, only 5 settings of μ_m lead to sufficiently feasible solutions returned by the variational technique.

Finally, a variable of interest to atmospheric scientists for winds inside MCSes is the vertical wind velocity. It provides a measure of the intensity of the storm by demonstrating the amount of upscale growth contributing to intensification. Figure 5 shows the mean updraft velocities inside the box in Figure 2 as a function of height for each of the runs of the TensorFlow L-BFGS-B and

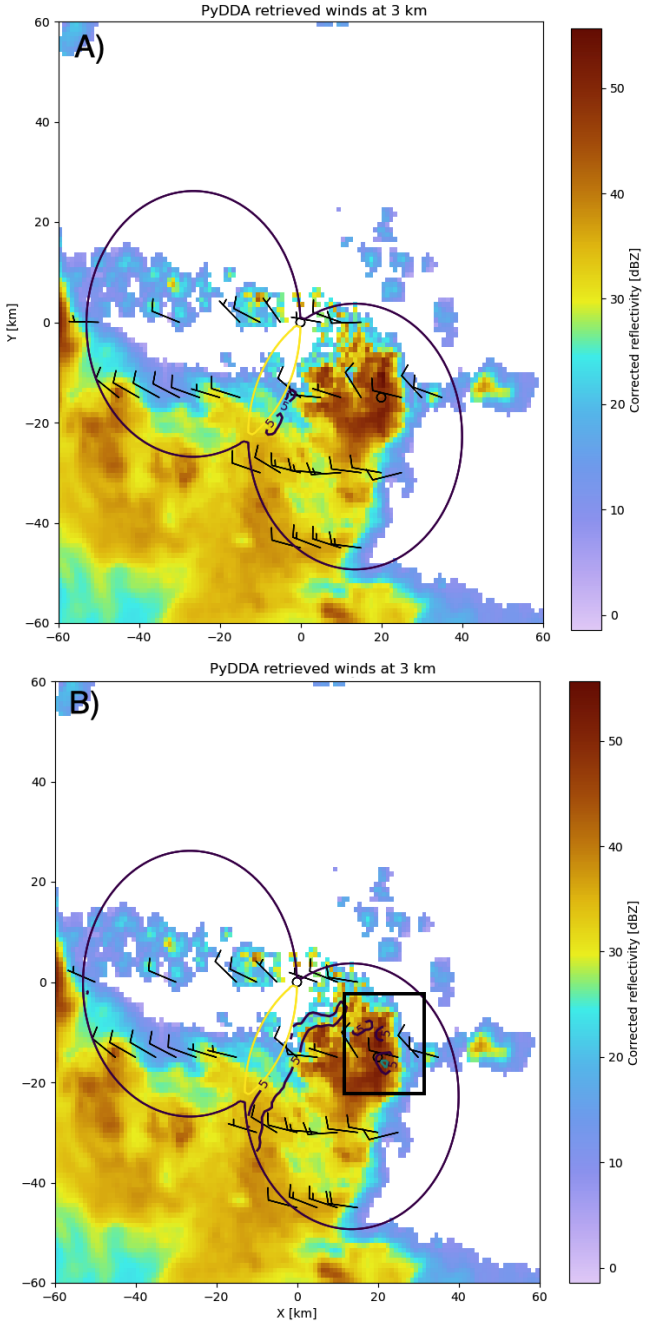


Fig. 2: *The PyDDA retrieved winds overlaid over reflectivity from the C-band Polarization Radar for the MCS that passed over Darwin, Australia on 20 Jan 2006. The winds were retrieved using the weak variational technique with $\mu = 1$ **(a)** and the Augmented Lagrangian technique with $\mu = 1$ **(b)**. The contours represent vertical velocities at 3.05 km altitude. The boxed region shows the updrafts that generated the heavy precipitation.*

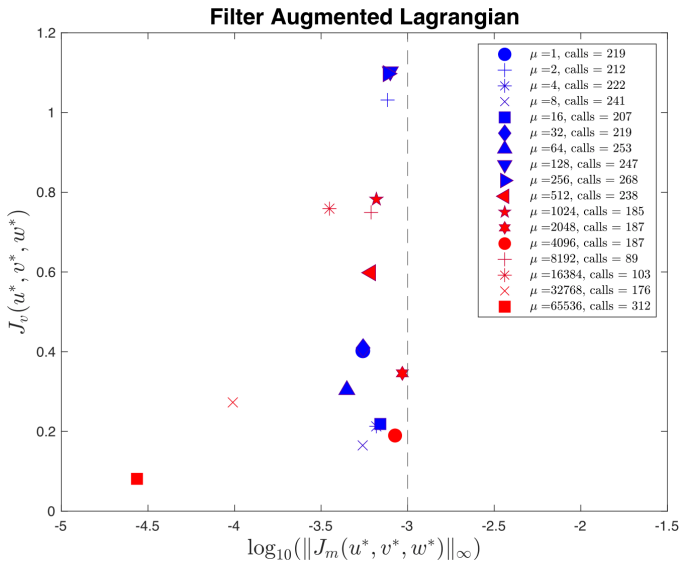


Fig. 3: *The x-axis shows, on a logarithmic scale, the maximum constraint violation in the units of divergence of the wind field and the y-axis shows the value of the data-fitting term J_v at the optimal solution. The legend lists the number of function/gradient calls made by the filter Augmented Lagrangian Method, which is the dominant cost of both approaches. The dashed line at 10^{-3} denotes the tolerance on the maximum constraint violation that was supplied to the filter Augmented Lagrangian method.*

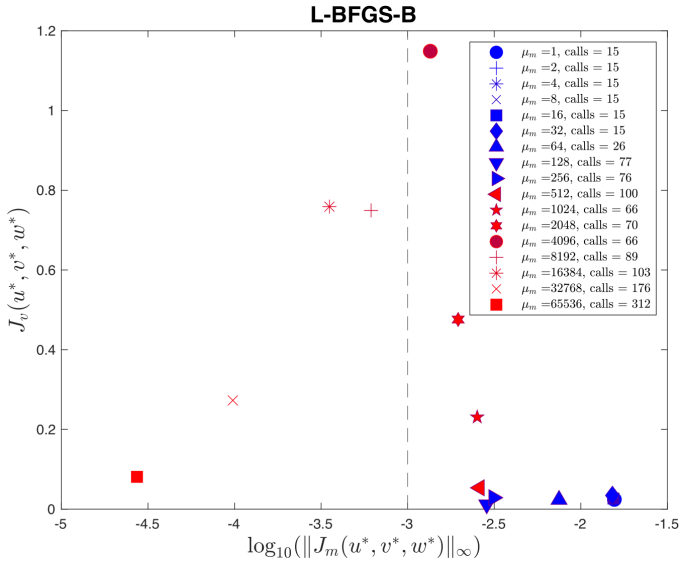


Fig. 4: *As 3, but for the weak variational technique that uses L-BFGS-B.*

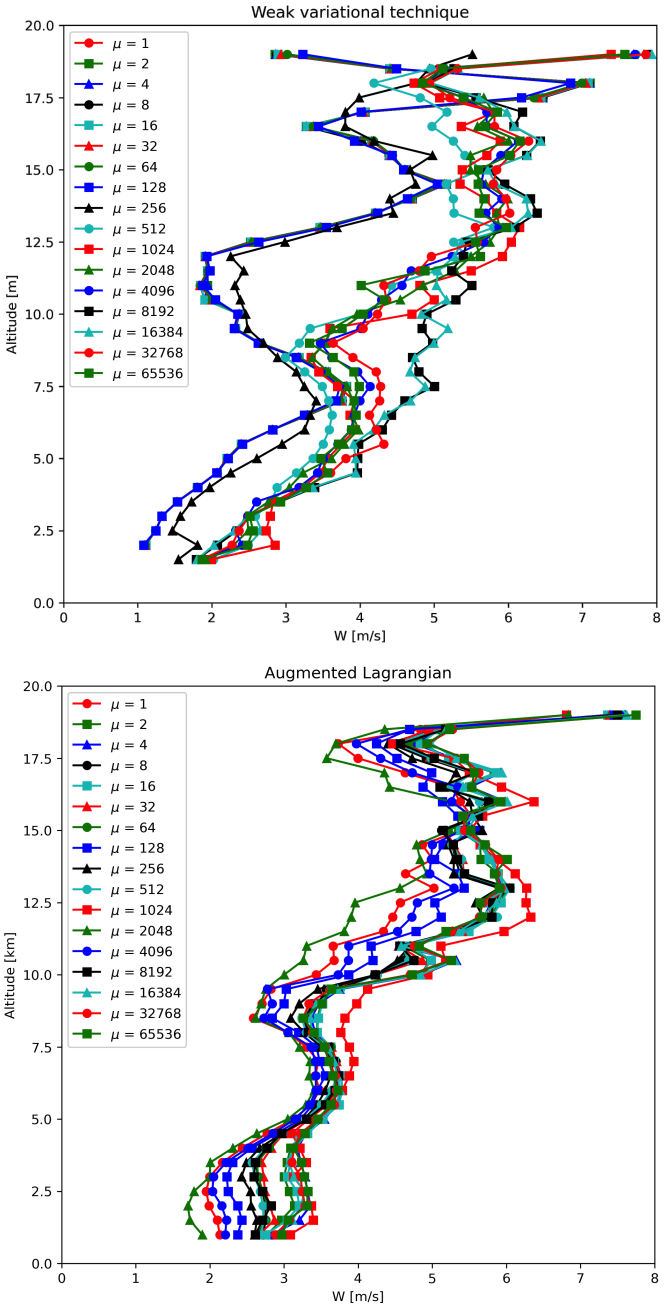


Fig. 5: *The mean updraft velocity obtained by (left) the weak variational and (right) the Augmented Lagrangian technique inside the updrafts in the boxed region of Figure 2. Each line represents a different value of μ for the given technique.*

Augmented Lagrangian techniques. Table 2 summarizes the mean and spread of the solutions in Figure 5. For the updraft velocities produced by the Augmented Lagrangian technique, there is a 1 m/s spread of velocities produced for given values of μ at altitudes < 7.5 km in Table 2. At an altitude of 10 km, this spread is 1.9 m/s. This is likely due to the reduced spatial coverage of the radars at higher altitudes. However, for the weak variational technique, the sensitivity of the retrieval to μ is much more pronounced, with up to 2.8 m/s differences between retrievals. Therefore, using the Augmented Lagrangian technique makes the vertical velocities less sensitive to μ. Therefore, this shows that using the Augmented Lagrangian technique will result in more reproducible wind fields from radar wind networks since it is less sensitive to user-defined parameters than the weak variational technique. However, a limitation of this technique is that, for now, this technique is limited to two radars and to the mass continuity and vertical vorticity constraints.

Concluding remarks

Atmospheric wind retrievals are vital for forecasting severe weather events. Therefore, this motivated us to develop an open source package for developing atmospheric wind retrievals called PyDDA. In the original releases of PyDDA (versions 0.5 and

	Min	Mean	Max	Std. Dev.
Weak variational				
2.5 km	1.2	1.8	2.7	0.6
5 km	2.2	2.9	4.0	0.7
7.5 km	3.2	3.9	5.0	0.4
10 km	2.3	3.3	4.9	1.0
Aug. Lagrangian				
2.5 km	1.8	2.8	3.3	0.5
5 km	3.1	3.3	3.5	0.1
7.5 km	3.2	3.5	3.9	0.1
10 km	3.0	4.3	4.9	0.5

TABLE 2: *Minimum, mean, maximum, and standard deviation of w (m/s) for select levels in Figure 5.*

prior), the original goal of PyDDA was to convert legacy wind retrieval packages such as CEDRIC and Multidop to be fully Pythonic, open source, and accessible to the scientific community. However, there remained many improvements to be made to PyDDA to optimize the speed of the retrievals and to make it easier to add constraints to PyDDA.

This therefore motivated two major changes to PyDDA's wind retrieval routine for PyDDA 1.0. The first major change to PyDDA in PyDDA 1.0 was to simplify the wind retrieval process by automating the calculation of the gradient of the cost function used for the weak variational technique. To do this, we utilized Jax and TensorFlow's capabilities to do automatic differentiation of functions. This also allows PyDDA to take advantage of GPU resources, significantly speeding up retrieval times for mesoscale retrievals at kilometer-scale resolution. In addition, running the TensorFlow-based version of PyDDA provided significant performance improvements even when using a CPU.

These automatically generated gradients were then used to implement an Augmented Lagrangian technique in PyDDA 1.1 that allows for automatically determining the weights for each cost function in the retrieval. The Augmented Lagrangian technique guarantees convergence to a physically realistic solution, something that is not always the case for a given set of weights for the weak variational technique. Therefore, this both creates more reproducible wind retrievals and simplifies the process of retrieving winds for the non-specialist user. However, since the Augmented Lagrangian technique currently only supports the ingesting of radar data into the retrieval, plans for PyDDA 1.2 and beyond include expanding the Augmented Lagrangian technique to support multiple data sources such as models and rawinsondes.

Code Availability

PyDDA is available for public use with documentation and examples available at https://openradarscience.org/PyDDA. The GitHub repository that hosts PyDDA's source code is available at https://github.com/openradar/PyDDA.

Acknowledgments

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ('Argonne'). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. This material is based upon work supported by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357. This material is also based upon work funded by program development funds from the Mathematics and Computer Science and Environmental Science departments at Argonne National Laboratory.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

[CPMW13] Scott Collis, Alain Protat, Peter T. May, and Christopher Williams. Statistics of storm updraft velocities from twp-ice including verification with profiling measurements. *Journal of Applied Meteorology and Climatology*, 52(8):1909 – 1922, 2013. doi:10.1175/JAMC-D-12-0230.1.

[FLT06] Roger Fletcher, Sven Leyffer, and Philippe Toint. A brief history of filter methods. Technical report, Argonne National Laboratory, 2006. URL: http://www.optimization-online.org/DB_FILE/2006/10/1489.pdf.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[JCL+18] R. C. Jackson, S. M. Collis, V. Louf, A. Protat, and L. Majewski. A 17 year climatology of the macrophysical properties of convection in darwin. *Atmospheric Chemistry and Physics*, 18(23):17687–17704, 2018. doi:10.5194/acp-18-17687-2018.

[JCL+20] Robert Jackson, Scott Collis, Timothy Lang, Corey Potvin, and Todd Munson. Pydda: A pythonic direct data assimilation framework for wind retrievals. *Journal of Open Research Software*, 8(1):20, 2020. doi:10.5334/jors.264.

[LN89] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *MATHEMATICAL PROGRAMMING*, 45:503–528, 1989. doi:10.1007/bf01589116.

[LSKJ17] Timothy Lang, Mario Souto, Shahin Khobahi, and Bobby Jackson. nasa/multidop: Multidop v0.3, October 2017. doi:10.5281/zenodo.1035904.

[LV20] Sven Leyffer and Charlie Vanaret. An augmented lagrangian filter method. *Mathematical Methods of Operations Research*, 92(2):343–376, 2020. URL: https://doi.org/10.1007/s00186-020-00713-x, doi:10.1007/s00186-020-00713-x.

[MF98] L. Jay Miller and Sherri M. Fredrick. Custom editing and display of reduced information in cartesian space. Technical report, National Center for Atmospheric Research, 1998.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[PSX12] Corey K. Potvin, Alan Shapiro, and Ming Xue. Impact of a vertical vorticity constraint in variational dual-doppler wind analysis: Tests with real and simulated supercell data. *Journal of Atmospheric and Oceanic Technology*, 29(1):32 – 49, 2012. doi:10.1175/JTECH-D-11-00019.1.

[RJSCTL+19] Robert Jackson, Scott Collis, Timothy Lang, Corey Potvin, and Todd Munson. PyDDA: A new Pythonic Wind Retrieval Package. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 111 – 117, 2019. doi:10.25080/Majora-7ddc1dd1-010.

[SPG09] Alan Shapiro, Corey K. Potvin, and Jidong Gao. Use of a vertical vorticity equation in variational dual-doppler wind analysis. *Journal of Atmospheric and Oceanic Technology*, 26(10):2089 – 2106, 2009. doi:10.1175/2009JTECHA1256.1.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

RocketPy: Combining Open-Source and Scientific Libraries to Make the Space Sector More Modern and Accessible

João Lemes Gribel Soares[‡*], Mateus Stano Junqueira[‡], Oscar Mauricio Prada Ramirez[‡], Patrick Sampaio dos Santos Brandão[‡§], Adriano Augusto Antongiovanni[‡], Guilherme Fernandes Alves[‡], Giovani Hidalgo Ceotto[‡]

Abstract—In recent years we are seeing exponential growth in the space sector, with new companies emerging in it. On top of that more people are becoming fascinated to participate in the aerospace revolution, which motivates students and hobbyists to build more High Powered and Sounding Rockets. However, rocketry is still a very inaccessible field, with high knowledge of entry-level and concrete terms. To make it more accessible, people need an active community with flexible, easy-to-use, and well-documented tools. RocketPy is a software solution created to address all those issues, solving the trajectory simulation for High-Power rockets being built on top of SciPy and the Python Scientific Environment. The code allows for a sophisticated 6 degrees of freedom simulation of a rocket's flight trajectory, including high fidelity variable mass effects as well as descent under parachutes. All of this is packaged into an architecture that facilitates complex simulations, such as multi-stage rockets, design and trajectory optimization, and dispersion analysis. In this work, the flexibility and usability of RocketPy are indicated in three example simulations: a basic trajectory simulation, a dynamic stability analysis, and a Monte Carlo dispersion simulation. The code structure and the main implemented methods are also presented.

Index Terms—rocketry, flight, rocket trajectory, flexibility, Monte Carlo analysis

Introduction

When it comes to rockets, there is a wide field ranging from orbital rockets to model rockets. Between them, two types of rockets are relevant to this work: sounding rockets and High-Powered Rockets (HPRs). Sounding rockets are mainly used by government agencies for scientific experiments in suborbital flights while HPRs are generally used for educational purposes, with increasing popularity in university competitions, such as the annual Spaceport America Cup, which hosts more than 100 rocket design teams from all over the world. After the university-built rocket TRAVELER IV [AEH+19] successfully reached space by crossing the Kármán line in 2019, both Sounding Rockets and HPRs can now be seen as two converging categories in terms of overall flight trajectory.

HPRs are becoming bigger and more robust, increasing their potential hazard, along with their capacity, making safety an important issue. Moreover, performance is always a requirement both for saving financial and time resources while efficiently launch performance goals.

In this scenario, crucial parameters should be determined before a safe launch can be performed. Examples include calculating with high accuracy and certainty the most likely impact or landing region. This information greatly increases range safety and the possibility of recovering the rocket [Wil18]. As another example, it is important to determine the altitude of the rocket's apogee in order to avoid collision with other aircraft and prevent airspace violations.

To better attend to those issues, RocketPy was created as a computational tool that can accurately predict all dynamic parameters involved in the flight of sounding, model, and High-Powered Rockets, given parameters such as the rocket geometry, motor characteristics, and environmental conditions. It is an open source project, well structured, and documented, allowing collaborators to contribute with new features with minimum effort regarding legacy code modification [CSA+21].

Background

Rocketry terminology

To better understand the current work, some specific terms regarding the rocketry field are stated below:

- Apogee: The point at which a body is furthest from earth
- Degrees of freedom: Maximum number of independent values in an equation
- Flight Trajectory: 3-dimensional path, over time, of the rocket during its flight
- Launch Rail: Guidance for the rocket to accelerate to a stable flight speed
- Powered Flight: Phase of the flight where the motor is active
- Free Flight: Phase of the flight where the motor is inactive and no other component but its inertia is influencing the rocket's trajectory
- Standard Atmosphere: Average pressure, temperature, and air density for various altitudes
- Nozzle: Part of the rocket's engine that accelerates the exhaust gases
- Static hot-fire test: Test to measure the integrity of the motor and determine its thrust curve

* *Corresponding author: jgribel@usp.br*
‡ *Escola Politécnica of the University of São Paulo*
§ *École Centrale de Nantes.*

- Thrust Curve: Evolution of thrust force generated by a motor
- Static Margin: Is a non-dimensional distance to analyze the stability
- Nosecone: The forward-most section of a rocket, shaped for aerodynamics
- Fin: Flattened append of the rocket providing stability during flight, keeping it in the flight trajectory

Flight Model

The flight model of a high-powered rocket takes into account at least three different phases:

1. The first phase consists of a linear movement along the launch rail: The motion of the rocket is restricted to one dimension, which means that only the translation along with the rail needs to be modeled. During this phase, four forces can act on the rocket: weight, engine thrust, rail reactions, and aerodynamic forces.

2. After completely leaving the rail, a phase of 6 degrees of freedom (DOF) is established, which includes powered flight and free flight: The rocket is free to move in three-dimensional space and weight, engine thrust, normal and axial aerodynamic forces are still important.

3. Once apogee is reached, a parachute is usually deployed, characterizing the third phase of flight: the parachute descent. In the last phase, the parachute is launched from the rocket, which is usually divided into two or more parts joined by ropes. This phase ends at the point of impact.

Design: RocketPy Architecture

Four main classes organize the dataflow during the simulations: motor, rocket, environment, and flight [CSA+21]. Furthermore, there is also a helper class named *function*, which will be described further. In the Motor class, the main physical and geometric parameters of the motor are configured, such as nozzle geometry, grain parameters, mass, inertia, and thrust curve. This first-class acts as an input to the Rocket class where the user is also asked to define certain parameters of the rocket such as the inertial mass tensor, geometry, drag coefficients, and parachute description. Finally, the Flight class joins the rocket and motor parameters with information from another class called Environment, such as wind, atmospheric, and earth models, to generate a simulation of the rocket's trajectory. This modular architecture, along with its well-structured and documented code, facilitates complex simulations, starting with the use of Jupyter Notebooks that people can adapt for their specific use case. Fig. 1 illustrates RocketPy architecture.

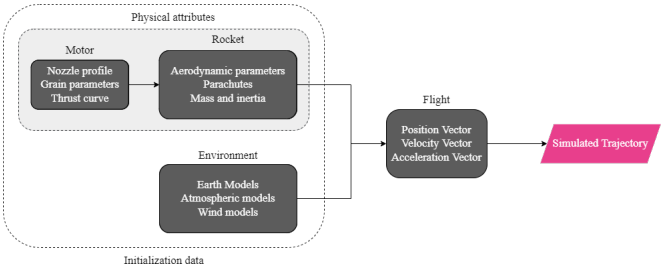


Fig. 1: *RocketPy classes interaction [CSA+21]*

Function

Variable interpolation meshes/grids from different sources can lead to problems regarding coupling different data types. To solve this, RocketPy employs a dedicated *Function* class which allows for more natural and dynamic handling of these objects, structuring them as $\mathbb{R}^n \to \mathbb{R}$ mathematical functions.

Through the use of those methods, this approach allows for quick and easy arithmetic operations between lambda expressions and list-defined interpolated functions, as well as scalars. Different interpolation methods are available to be chosen from, among them simple polynomial, spline, and Akima ([Aki70]). Extrapolation of *Function* objects outside the domain constrained by a given dataset is also allowed.

Furthermore, evaluation of definite integrals of these *Function* objects is among their feature set. By cleverly exploiting the chosen interpolation option, RocketPy calculates the values fast and precisely through the use of different analytical methods. If numerical integration is required, the class makes use of SciPy's implementation of the QUADPACK Fortran library [PdDKÜK83]. For 1-dimensional Functions, evaluation of derivatives at a point is made possible through the employment of a simple finite difference method.

Finally, to increase usability and readability, all *Function* object instances are callable and can be presented in multiple ways depending on the given arguments. If no argument is given, a Matplotlib figure opens and the plot of the function is shown inside its domain. Only 2-dimensional and 3-dimensional functions can be plotted. This is especially useful for the post-processing methods where various information on the classes responsible for the definition of the rocket and its flight is presented, providing for more concise code. If an n-sized array is passed instead, RocketPy will try and evaluate the value of the Function at this given point using different methods, returning its value. An example of the usage of the Function class can be found in the Examples section.

Additionally, if another *Function* object is passed, the class will try to match their respective domain and co-domain in order to return a third instance, representing a composition of functions, in the likes of: $h(x) = (g \circ f)(x) = g(f(x))$. With different *Function* objects defined, the *comparePlots* method can be used to plot, in a single graph, different functions.

By imitating, in syntax, commonly used mathematical notation, RocketPy allows for more understandable and human-readable code, especially in the implementation of the more extensive and cluttered rocket equations of motion.

Environment

The Environment class reads, processes and stores all the information regarding wind and atmospheric model data. It receives as inputs launch point coordinates, as well as the length of the launch rail, and then provides the flight class with six profiles as a function of altitude: wind speed in east and north directions, atmospheric pressure, air density, dynamic viscosity, and speed of sound. For instance, an Environment object can be set as representing New Mexico, United States:

```
from rocketpy import Environment

ex_env = Environment(
    railLength=5.2,
    latitude=32.990254,
    longitude=-106.974998,
    elevation=1400
)
```

RocketPy requires *datetime* library information specifying the year, month, day and hour to compute the weather conditions on the specified day of launch. An optional argument, the timezone, may also be specified. If the user prefers to omit it, RocketPy will assume the *datetime* object is given in standard UTC time, just as follows:

```
import datetime
tomorrow = (
    datetime.date.today() +
    datetime.timedelta(days=1)
)

date_info = (
    tomorrow.year,
    tomorrow.month,
    tomorrow.day,
    12
)  # Hour given in UTC time
```

By default, the International Standard Atmosphere [ISO75] static atmospheric model is loaded. However, it is easy to set other models by importing data from different meteorological agencys' public datasets, such as Wyoming Upper-Air Soundings and European Centre for Medium-Range Weather Forecasts (ECMWF); or to set a customized atmospheric model based on user-defined functions. As RocketPy supports integration with different meteorological agencies' datasets, it allows for a sophisticated definition of weather conditions including forecasts and historical reanalysis scenarios.

In this case, NOAA's RUC Soundings data model is used, a worldwide and open-source meteorological model made available online. The file name is set as *GFS*, indicating the use of the Global Forecast System provided by NOAA, which features a forecast with a quarter degree equally spaced longitude/latitude grid with a temporal resolution of three hours.

```
ex_env.setAtmosphericModel(
    type='Forecast',
    file='GFS')
ex_env.info()
```

What is happening on the back-end of this code's snippet is RocketPy utilizing the OPeNDAP protocol to retrieve data arrays from NOAA's server. It parses by using the netCDF4 data management system, allowing for the retrieval of pressure, temperature, wind velocity, and surface elevation data as a function of altitude. The Environment class then computes the following parameters: wind speed, wind heading, speed of sound, air density, and dynamic viscosity. Finally, plots of the evaluated parameters concerning the altitude are all passed on to the mission analyst by calling the *Env.info()* method.

Motor

RocketPy is flexible enough to work with most types of motors used in sound rockets. The main function of the Motor class is to provide the thrust curve, the propulsive mass, the inertia tensor, and the position of its center of mass as a function of time. Geometric parameters regarding propellant grains and the motor's nozzle must be provided, as well as a thrust curve as a function of time. The latter is preferably obtained empirically from a static hot-fire test, however, many of the curves for commercial motors are freely available online [Cok98].

Alternatively, for homemade motors, there is a wide range of open-source internal ballistics simulators, such as OpenMotor [Rei22], can predict the produced thrust with high accuracy for a given sizing and propellant combination. There are different types of rocket motors: solid motors, liquid motors, and hybrid motors. Currently, a robust Solid Motor class has been fully implemented and tested. For example, a typical solid motor can be created as an object in the following way:

```
from rocketpy import SolidMotor

ex_motor = SolidMotor(
    thrustSource='Motor_file.eng',
    burnOut=2,
    reshapeThrustCurve= False,
    grainNumber=5,
    grainSeparation=3/1000,
    grainOuterRadius=33/1000,
    grainInitialInnerRadius=15/1000,
    grainInitialHeight=120/1000,
    grainDensity= 1782.51,
    nozzleRadius=49.5/2000,
    throatRadius=21.5/2000,
    interpolationMethod='linear')
```

Rocket

The Rocket Class is responsible for creating and defining the rocket's core characteristics. Mostly composed of physical attributes, such as mass and moments of inertia, the rocket object will be responsible for storage and calculate mechanical parameters.

A rocket object can be defined with the following code:

```
from rocketpy import Rocket

ex_rocket = Rocket(
    motor=ex_motor,
    radius=127 / 2000,
    mass=19.197 - 2.956,
    inertiaI=6.60,
    inertiaZ=0.0351,
    distanceRocketNozzle=-1.255,
    distanceRocketPropellant=-0.85704,
    powerOffDrag="data/rocket/powerOffDragCurve.csv",
    powerOnDrag="data/rocket/powerOnDragCurve.csv",
)
```

As stated in [RocketPy architecture], a fundamental input of the rocket is its motor, an object of the Motor class that must be previously defined. Some inputs are fairly simple and can be easily obtained with a CAD model of the rocket such as radius, mass, and moment of inertia on two different axes. The *distance* inputs are relative to the center of mass and define the position of the motor nozzle and the center of mass of the motor propellant. The *powerOffDrag* and *powerOnDrag* receive .csv data that represents the drag coefficient as a function of rocket speed for the case where the motor is off and other for the motor still burning, respectively.

At this point, the simulation would run a rocket with a tube of a certain diameter, with its center of mass specified and a motor at its end. For a better simulation, a few more important aspects should then be defined, called *Aerodynamic surfaces*. Three of them are accepted in the code, these being the nosecone, fins, and tail. They can be simply added to the code via the following methods:

```
nose_cone = ex_rocket.addNose(
    length=0.55829, kind="vonKarman",
    distanceToCM=0.71971
)
fin_set = ex_rocket.addFins(
    4, span=0.100, rootChord=0.120, tipChord=0.040,
    distanceToCM=-1.04956
)
tail = ex_rocket.addTail(
    topRadius=0.0635, bottomRadius=0.0435,
    length=0.06, distanceToCM=-1.194656
)
```

All these methods receive defining geometrical parameters and their distance to the rocket's center of mass (distanceToCM) as inputs. Each of these surfaces generates, during the flight, a lift force that can be calculated via a lift coefficient, which is calculated with geometrical properties, as shown in [Bar67]. Further on, these coefficients are used to calculate the center of pressure and subsequently the static margin. In each of these methods, the static margin is reevaluated.

Finally, the parachutes can be added in a similar manner to the aerodynamic surfaces. However, a few inputs regarding the electronics involved in the activation of the parachute are required. The most interesting of them is the *trigger* and *samplingRate* inputs, which are used to define the parachute's activation. The *trigger* is a function that returns a boolean value that signifies when the parachute should be activated. The *samplingRate* is the time interval that the *trigger* will be evaluated in the simulation time steps.

```
def parachute_trigger(p, y):
    if vel_z < 0 and height < 800:
        boole = True
    else:
        boole = False
    return boole

ex_parachute = ex_rocket.addParachute(
    'ParachuteName',
    CdS=10.0,
    trigger=parachute_trigger,
    samplingRate=105,
    lag=1.5,
    noise=(0, 8.3, 0.5)
)
```

With the rocket fully defined, the `Rocket.info()` and `Rocket.allInfo()` methods can be called giving us information and plots of the calculations performed in the class. One of the most relevant outputs of the Rocket class is the static margin, as it is important for the rocket stability and makes possible several analyses. It is visualized through the time plot in Fig. 2, which shows the variation of the static margin as the motor burns its propellant.

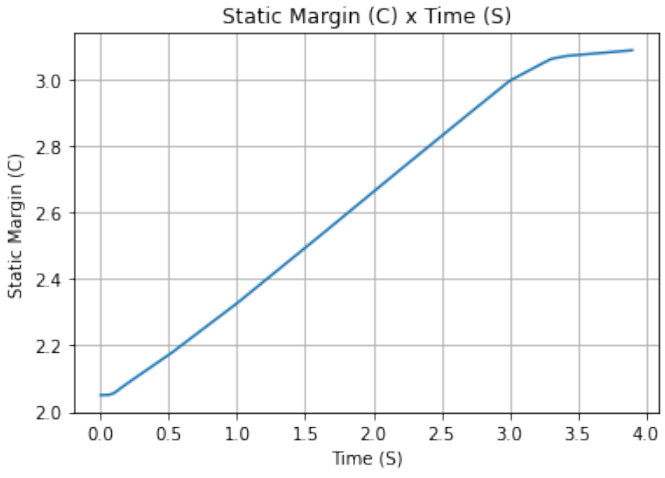


Fig. 2: *Static Margin*

Flight

The Flight class is responsible for the integration of the rocket's equations of motion overtime [CSA+21]. Data from instances of the Rocket class and the Environment class are used as input to initialize it, along with parameters such as launch heading and inclination relative to the Earth's surface:

```
from rocketpy import Flight

ex_flight = Flight(
    rocket=rocket,
    environment=env,
    inclination=85,
    heading=0
)
```

Once the simulation is initialized, run, and completed, the instance of the Flight class stores relevant raw data. The `Flight.postProcess()` method can then be used to compute secondary parameters such as the rocket's Mach number during flight and its angle of attack.

To perform the numerical integration of the equations of motion, the Flight class uses the LSODA solver [Pet83] implemented by Scipy's `scipy.integrate` module [VGO+20]. Usually, well-designed rockets result in non-stiff equations of motion. However, during flight, rockets may become unstable due to variations in their inertial and aerodynamic properties, which can result in a stiff system. LSODA switches automatically between the nonstiff Adams method and the stiff BDF method, depending on the detected stiffness, perfectly handle both cases.

Since a rocket's flight trajectory is composed of multiple phases, each with its own set of governing equations, RocketPy employs a couple of clever methods to run the numerical integration. The Flight class uses a `FlightPhases` container to hold each `FlightPhase`. The `FlightPhases` container will orchestrate the different `FlightPhase` instances, and compose them during the flight.

This is crucial because there are events that may or may not happen during the simulation, such as the triggering of a parachute ejection system (which may or may not fail) or the activation of a premature flight termination event. There are also events such as the departure from the launch rail or the apogee that is known to occur, but their timestamp is unknown until the simulation is run. All of these events can trigger new flight phases, characterized by a change in the rocket's equations of motion. Furthermore, such events can happen close to each other and provoke delayed phases.

To handle this, the Flight class has a mechanism for creating new phases and adding them dynamically in the appropriate order to the `FlightPhases` container.

The constructor of the `FlightPhase` class takes the following arguments:

- `t`: a timestamp that symbolizes at which instant such flight phase should begin;
- `derivative`: a function that returns the time derivatives of the rocket's state vector (i.e., calculates the equations of motion for this flight phase);
- `callbacks`: a list of callback functions to be run when the flight phase begins (which can be useful if some parameters of the rocket need to be modified before the flight phase begins).

The constructor of the Flight class initializes the `FlightPhases` container with a *rail phase* and also a dummy *max time* phase which marks the maximum flight duration. Then, it loops through the elements of the container.

Inside the loop, an important attribute of the current flight phase is set: `FlightPhase.timeBound`, the maxi-

mum timestamp of the flight phase, which is always equal to the initial timestamp of the next flight phase. Ordinarily, it would be possible to run the LSODA solver from `FlightPhase.t` to `FlightPhase.timeBound`. However, this is not an option because the events which can trigger new flight phases need to be checked throughout the simulation. While `scipy.integrate.solve_ivp` does offer the `events` argument to aid in this, it is not possible to use it with most of the events that need to be tracked, since they cannot be expressed in the necessary form.

As an example, consider the very common event of a parachute ejection system. To simulate real-time algorithms, the necessary inputs to the ejection algorithm need to be supplied at regular intervals to simulate the desired sampling rate. Furthermore, the ejection algorithm cannot be called multiple times without real data since it generally stores all the inputs it gets to calculate if the rocket has reached the apogee to trigger the parachute release mechanism. Discrete controllers can present the same peculiar properties.

To handle this, the instance of the `FlightPhase` class holds a `TimeNodes` container, which stores all the required timesteps, or `TimeNode`, that the integration algorithm should stop at so that the events can be checked, usually by feeding the necessary data to parachutes and discrete control trigger functions. When it comes to discrete controllers, they may change some parameters in the rocket once they are called. On the other hand, a parachute triggers rarely actually trigger, and thus, rarely invoke the creation of a new flight phase characterized by *descent under parachute* governing equations of motion.

The Flight class can take advantage of this fact by employing overshootable time nodes: time nodes that the integrator does not need to stop. This allows the integration algorithm to use more optimized timesteps and significantly reduce the number of iterations needed to perform a simulation. Once a new timestep is taken, the Flight class checks all overshootable time nodes that have passed and feeds their event triggers with interpolated data. In case when an event is triggered, the simulation is rolled back to that state.

In summary, throughout a simulation, the Flight class loops through each non-overshootable `TimeNode` of each element of the `FlightPhases` container. At each `TimeNode`, the event triggers are fed with the necessary input data. Once an event is triggered, a new `FlightPhase` is created and added to the main container. These loops continue until the simulation is completed, either by reaching the maximum flight duration or by reaching a terminal event, such as ground impact.

Once the simulation is completed, raw data can already be accessed. To compute secondary parameters, the `Flight.postProcess()` is used. It takes advantage of the fact that the `FlightPhases` container keeps all relevant flight information to essentially retrace the trajectory and capture more information about the flight.

Once secondary parameters are computed, the `Flight.allInfo` method can be used to show and plot all the relevant information, as illustrated in Fig. 3.

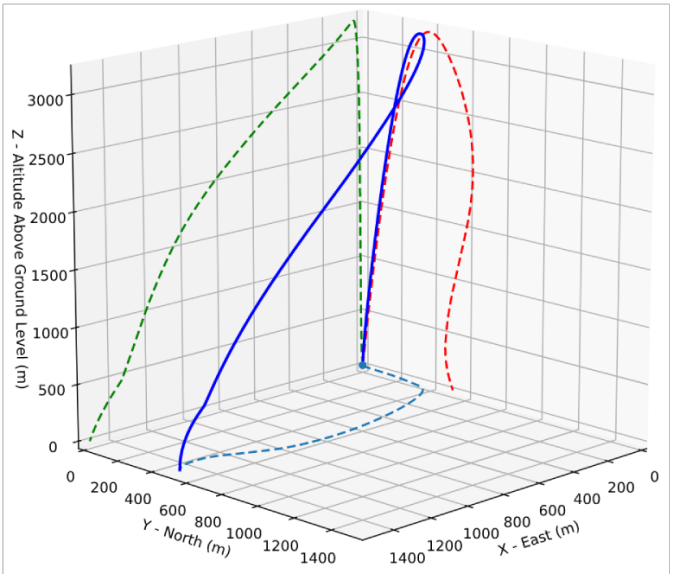


Fig. 3: *3D flight trajectory, an output of the Flight.allInfo method*

The adaptability of the Code and Accessibility

RocketPy's development started in 2017, and since the beginning, certain requirements were kept in mind:

- Execution times should be **fast**. There is a high interest in performing sensitivity analysis, optimization studies and Monte Carlo simulations, which require a large number of simulations to be performed (10,000 ~ 100,000).
- The code structure should be **flexible**. This is important due to the diversity of possible scenarios that exist in a rocket design context. Each user will have their simulation requirements and should be able to modify and adapt new features to meet their needs. For this reason, the code was designed in a fashion such that each major component is separated into self-encapsulated classes, responsible for a single functionality. This tenet follows the concepts of the so-called Single Responsibility Principle (SRP) [MNK03].
- Finally, the software should aim to be **accessible**. The source code was openly published on GitHub (https://github.com/Projeto-Jupiter/RocketPy), where the community started to be built and a group of developers, known as the RocketPy Team, are currently assigned as dedicated maintainers. The job involves not only helping to improve the code, but also working towards building a healthy ecosystem of Python, rocketry, and scientific computing enthusiasts alike; thus facilitating access to the high-quality simulation without a great level of specialization.

The following examples demonstrate how RocketPy can be a useful tool during the design and operation of a rocket model, enabling functionalities not available by other simulation software before.

Examples

Using RocketPy for Rocket Design

1) Apogee by Mass using a Function helper class

Because of performance and safety reasons, apogee is one of the most important results in rocketry competitions, and it's highly valuable for teams to understand how different Rocket parameters can change it. Since a direct relation is not available for this kind of computation, the characteristic of running simulation quickly is utilized for evaluation of how the Apogee is affected by the mass of the Rocket. This function is highly used during the early phases of the design of a Rocket.

An example of code of how this could be achieved:

```
from rocketpy import Function

def apogee(mass):
    # Prepare Environment
    ex_env = Environment(...)

    ex_env.setAtmosphericModel(
        type="CustomAtmosphere",
        wind_v=-5
    )

    # Prepare Motor
    ex_motor = SolidMotor(...)

    # Prepare Rocket
    ex_rocket = Rocket(
        ...,
        mass=mass,
        ...
    )

    ex_rocket.setRailButtons([0.2, -0.5])
    nose_cone = ex_rocket.addNose(.....)
    fin_set = ex_rocket.addFins(....)
    tail = ex_rocket.addTail(....)

    # Simulate Flight until Apogee
    ex_flight = Flight(.....)
    return ex_flight.apogee

apogee_by_mass = Function(
    apogee, inputs="Mass (kg)",
    outputs="Estimated Apogee (m)"
)
apogee_by_mass.plot(8, 20, 20)
```

The possibility of generating this relation between mass and apogee in a graph shows the flexibility of Rocketpy and also the importance of the simulation being designed to run fast.

1) Dynamic Stability Analysis

In this analysis the integration of three different RocketPy classes will be explored: Function, Rocket, and Flight. The motivation is to investigate how static stability translates into dynamic stability, i.e. different static margins result relies on different dynamic behavior, which also depends on the rocket's rotational inertia.

We can assume the objects stated in [motor] and [rocket] sections and just add a couple of variations on some input data to visualize the output effects. More specifically, the idea will be to explore how the dynamic stability of the studied rocket varies by changing the position of the set of fins by a certain factor.

To do that, we have to simulate multiple flights with different static margins, which is achieved by varying the rocket's fin positions. This can be done through a simple python loop, as described below:

```
simulation_results = []
for factor in [0.5, 0.7, 0.9, 1.1, 1.3]:
    # remove previous fin set
    ex_rocket.aerodynamicSurfaces.remove(fin_set)
    fin_set = ex_rocket.addFins(
        4, span=0.1, rootChord=0.120, tipChord=0.040,
        distanceToCM=-1.04956 * factor
    )
    ex_flight = Flight(
        rocket=ex_rocket,
        environment=env,
        inclination=90,
        heading=0,
        maxTimeStep=0.01,
        maxTime=5,
        terminateOnApogee=True,
        verbose=True,
    )
    ex_flight.postProcess()
    simulation_results += [(
        ex_flight.attitudeAngle,
        ex_rocket.staticMargin(0),
        ex_rocket.staticMargin(ex_flight.outOfRailTime),
        ex_rocket.staticMargin(ex_flight.tFinal)
    )]
Function.comparePlots(
    simulation_results,
    xlabel="Time (s)",
    ylabel="Attitude Angle (deg)",
)
```

The next step is to start the simulations themselves, which can be done through a loop where the Flight class is called, perform the simulation, save the desired parameters into a list and then follow through with the next iteration. The *post-process* flight data method is being used to make RocketPy evaluate additional result parameters after the simulation.

Finally, the *Function.comparePlots()* method is used to plot the final result, as reported at Fig. 4.

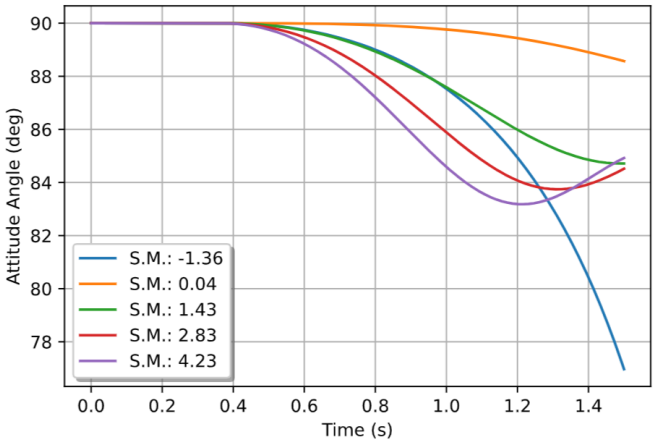


Fig. 4: *Dynamic Stability example, unstable rocket presented on blue line*

Monte Carlo Simulation

When simulating a rocket's trajectory, many input parameters may not be completely reliable due to several uncertainties in measurements raised during the design or construction phase of the rocket. These uncertainties can be considered together in a group of Monte Carlo simulations [RK16] which can be built on top of RocketPy.

The Monte Carlo method here is applied by running a significant number of simulations where each iteration has a different set of inputs that are randomly sampled given a previously known probability distribution, for instance the mean and standard deviation of a Gaussian distribution. Almost every input data presents some kind of uncertainty, except for the number of fins or propellant grains that a rocket presents. Moreover, some inputs, such as wind conditions, system failures, or the aerodynamic coefficient curves, may behave differently and must receive special treatment.

Statistical analysis can then be made on all the simulations, with the main result being the 1σ, 2σ, and 3σ ellipses representing the possible area of impact and the area where the apogee is

reached (Fig. 5). All ellipses can be evaluated based on the method presented by [Che66].

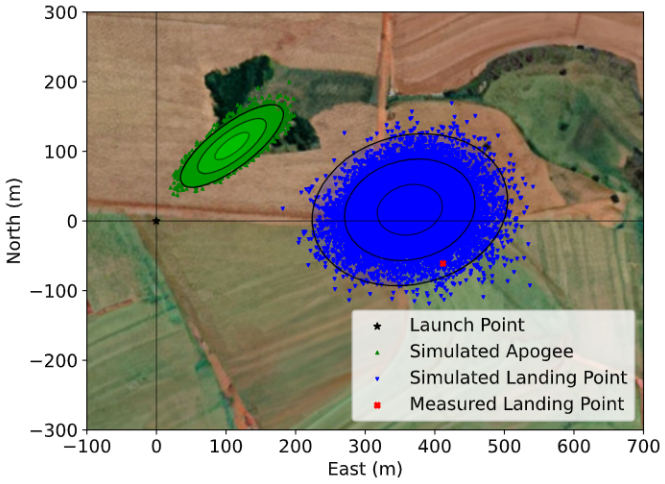


Fig. 5: *1* 1σ*, 2* 2σ*, and 3* 3σ *dispersion ellipses for both apogee and landing point*

When performing the Monte Carlo simulations on RocketPy, all the inputs - i.e. the parameters along with their respective standard deviations - are stored in a dictionary. The randomized set of inputs is then generated using a *yield* function:

```
def sim_settings(analysis_params, iter_number):
    i = 0
    while i < iter_number:
        # Generate a simulation setting
        sim_setting = {}
        for p_key, p_value in analysis_params.items():
            if type(p_value) is tuple:
                sim_setting[p_key] = normal(*p_value)
            else:
                sim_setting[p_key] = choice(p_value)
        # Update counter
        i += 1
        # Yield a simulation setting
        yield sim_setting
```

Where *analysis_params* is the dictionary with the inputs and *iter_number* is the total number of simulations to be performed. At that time the function yields one dictionary with one set of inputs, which will be used to run a simulation. Later the *sim_settings* function is called again and another simulation is run until the loop iterations reach the number of simulations:

```
for s in sim_settings(analysis_params, iter_number):
    # Define all classes to simulate with the current
    # set of inputs generated by sim_settings

    # Prepare Environment
    ex_env = Environment(.....)
    # Prepare Motor
    ex_motor = SolidMotor(.....)
    # Prepare Rocket
    ex_rocket = Rocket(.....)
    nose_cone = ex_rocket.addNose(.....)
    fin_set = ex_rocket.addFins(....)
    tail = ex_rocket.addTail(.....)

    # Considers any possible errors in the simulation
    try:
      # Simulate Flight until Apogee
      ex_flight = Flight(.....)

      # Function to export all output and input
      # data to a text file (.txt)
      export_flight_data(s, ex_flight)
    except Exception as E:
      # if an error occurs, export the error
      # message to a text file
      print(E)
      export_flight_error(s)
```

Finally, the set of inputs for each simulation along with its set of outputs, are stored in a .txt file. This allows for long-term data storage and the possibility to append simulations to previously finished ones. The stored output data can be used to study the final probability distribution of key parameters, as illustrated on Fig. 6.

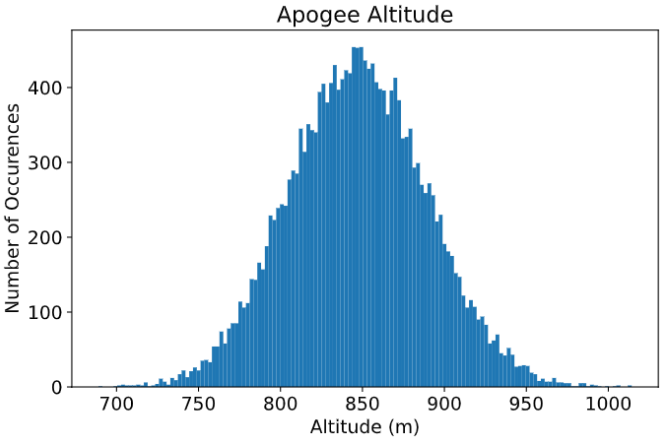


Fig. 6: *Distribution of apogee altitude*

Finally, it is also worth mentioning that all the information generated in the Monte Carlo simulation is based on RocketPy may be of utmost importance to safety and operational management during rocket launches, once it allows for a more reliable prediction of the landing site and apogee coordinates.

Validation of the results: Unit, Dimensionality and Acceptance Tests

Validation is a big problem for libraries like RocketPy, where true values for some results like apogee and maximum velocity is very hard to obtain or simply not available. Therefore, in order to make RocketPy more robust and easier to modify, while maintaining precise results, some innovative testing strategies have been implemented.

First of all, unit tests were implemented for all classes and their methods ensuring that each function is working properly. Given a set of different inputs that each function can receive, the respective outputs are tested against expected results, which can be based on real data or augmented examples cases. The test fails if the output deviates considerably from the established conditions, or an unexpected error occurs along the way.

Since RocketPy relies heavily on mathematical functions to express the governing equations, implementation errors can occur due to the convoluted nature of such expressions. Hence, to reduce the probability of such errors, there is a second layer of testing which will evaluate if such equations are dimensionally correct.

To accomplish this, RocketPy makes use of the *numericalunits* library, which defines a set of independent base units as randomly-chosen positive floating point numbers. In a dimensionally-correct function, the units all cancel out when the final answer is divided by its resulting unit. And thus, the result is deterministic, not

random. On the other hand, if the function contains dimensionally-incorrect equations, there will be random factors causing a randomly-varying final answer. In practice, RocketPy runs two calculations: one without *numericalunits*, and another with the dimensionality variables. The results are then compared to assess if the dimensionality is correct.

Here is an example. First, a SolidMotor object and a Rocket object are initialized without *numericalunits*:

```
@pytest.fixture
def unitless_solid_motor():
    return SolidMotor(
        thrustSource="Cesaroni_M1670.eng",
        burnOut=3.9,
        grainNumber=5,
        grainSeparation=0.005,
        grainDensity=1815,
        ...
    )

@pytest.fixture
def unitless_rocket(solid_motor):
    return Rocket(
        motor=unitless_solid_motor,
        radius=0.0635,
        mass=16.241,
        inertiaI=6.60,
        inertiaZ=0.0351,
        distanceRocketNozzle=-1.255,
        distanceRocketPropellant=-0.85704,
        ...
    )
```

Then, a SolidMotor object and a Rocket object are initialized with *numericalunits*:

```
import numericalunits

@pytest.fixture
def m():
    return numericalunits.m


@pytest.fixture
def kg():
    return numericalunits.kg

@pytest.fixture
def unitful_motor(kg, m):
    return SolidMotor(
        thrustSource="Cesaroni_M1670.eng",
        burnOut=3.9,
        grainNumber=5,
        grainSeparation=0.005 * m,
        grainDensity=1815 * (kg / m**3),
        ...
    )

@pytest.fixture
def unitful_rocket(kg, m, dimensionless_motor):
    return Rocket(
        motor=unitful_motor,
        radius=0.0635 * m,
        mass=16.241 * kg,
        inertiaI=6.60 * (kg * m**2),
        inertiaZ=0.0351 * (kg * m**2),
        distanceRocketNozzle=-1.255 * m,
        distanceRocketPropellant=-0.85704 * m,
        ...
    )
```

Then, to ensure that the equations implemented in both classes (`Rocket` and `SolidMotor`) are dimensionally correct, the values computed can be compared. For example, the `Rocket` class computes the rocket's static margin, which is a non-dimensional value and the result from both calculations should be the same:

```
def test_static_margin_dimension(
    unitless_rocket,
    unitful_rocket
):
    ...
    s1 = unitless_rocket.staticMargin(0)
    s2 = unitful_rocket.staticMargin(0)
    assert abs(s1 - s2) < 1e-6
```

In case the value of interest has units, such as the position of the center of pressure of the rocket, which has units of length, then such value must be divided by the relevant unit for comparison:

```
def test_cp_position_dimension(
    unitless_rocket,
    unitful_rocket
):
    ...
    cp1 = unitless_rocket.cpPosition(0)
    cp2 = unitful_rocket.cpPosition(0) / m
    assert abs(cp1 - cp2) < 1e-6
```

If the assertion fails, we can assume that the formula responsible for calculating the center of pressure position was implemented incorrectly, probably with a dimensional error.

Finally, some tests at a larger scale, known as acceptance tests, were implemented to validate outcomes such as apogee, apogee time, maximum velocity, and maximum acceleration when compared to real flight data. A required accuracy for such values were established after the publication of the experimental data by [CSA+21]. Such tests are crucial for ensuring that the code doesn't lose precision as a result of new updates.

These three layers of testing ensure that the code is trustworthy, and that new features can be implemented without degrading the results.

Conclusions

RocketPy is an easy-to-use tool for simulating high-powered rocket trajectories built with SciPy and the Python Scientific Environment. The software's modular architecture is based on four main classes and helper classes with well-documented code that allows to easily adapt complex simulations to various needs using the supplied Jupyter Notebooks. The code can be a useful tool during Rocket design and operation, allowing to calculate of key parameters such as apogee and dynamic stability as well as high-fidelity 6-DOF vehicle trajectory with a wide variety of customizable parameters, from its launch to its point of impact. RocketPy is an ever-evolving framework and is also accessible to anyone interested, with an active community maintaining it and working on future features such as the implementation of other engine types, such as hybrids and liquids motors, and even orbital flights.

Installing RocketPy

RocketPy was made to run on Python 3.6+ and requires the packages: Numpy >=1.0, Scipy >=1.0 and Matplotlib >= 3.0. For a complete experience we also recommend netCDF4 >= 1.4. All these packages, except netCDF4, will be installed automatically if the user does not have them. To install, execute:

```
pip install rocketpy
```

or

```
conda install -c conda-forge rocketpy
```

The source code, documentation and more examples are available at https://github.com/Projeto-Jupiter/RocketPy

Acknowledgments

The authors would like to thank the *University of São Paulo*, for the support during the development of the current publication, and also all members of Projeto Jupiter and the RocketPy Team who contributed to the making of the RocketPy library.

REFERENCES

[AEH+19] Adam Aitoumeziane, Peter Eusebio, Conor Hayes, Vivek Ramachandran, Jamie Smith, Jayasurya Sridharan, Luke St Regis, Mark Stephenson, Neil Tewksbury, Madeleine Tran, and Haonan Yang. Traveler IV Apogee Analysis. Technical report, USC Rocket Propulsion Laboratory, Los Angeles, 2019. URL: http://www.uscrpl.com/s/Traveler-IV-Whitepaper.

[Aki70] Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of the ACM (JACM)*, 17(4):589–602, 1970. doi:10.1145/321607.321609.

[Bar67] James S Barrowman. *The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles*. PhD thesis, Catholic University of America, Washington, DC United States, 1967.

[Che66] Victor Chew. Confidence, Prediction, and Tolerance Regions for the Multivariate Normal Distribution. *Journal of the American Statistical Association*, 61(315), 1966. doi:10.1080/01621459.1966.10480892.

[Cok98] J Coker. Thrustcurve.org — rocket motor performance data online, 1998. URL: https://www.thrustcurve.org/.

[CSA+21] Giovani H Ceotto, Rodrigo N Schmitt, Guilherme F Alves, Lucas A Pezente, and Bruno S Carmo. Rocketpy: Six degree-of-freedom rocket trajectory simulator. *Journal of Aerospace Engineering*, 34(6), 2021. doi:10.1061/(ASCE)AS.1943-5525.0001331.

[ISO75] ISO Central Secretary. Standard Atmosphere. Technical Report ISO 2533:1975, International Organization for Standardization, Geneva, CH, 5 1975.

[MNK03] Robert C Martin, James Newkirk, and Robert S Koss. *Agile software development: principles, patterns, and practices*, volume 2. Prentice Hall Upper Saddle River, NJ, 2003.

[PdDKÜK83] Robert Piessens, Elise de Doncker-Kapenga, Christoph W Überhuber, and David K Kahaner. *Quadpack: a subroutine package for automatic integration*, volume 1. Springer Science & Business Media, 1983. doi:10.1007/978-3-642-61786-7.

[Pet83] Linda Petzold. Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. *SIAM Journal on Scientific and Statistical Computing*, 4(1):136–148, 3 1983. doi:10.1137/0904010.

[Rei22] A Reilley. openmotor: An open-source internal ballistics simulator for rocket motor experimenters, 2022. URL: https://github.com/reilleya/openMotor.

[RK16] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016. doi:10.1002/9781118631980.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[Wil18] Paul D. Wilde. Range safety requirements and methods for sounding rocket launches. *Journal of Space Safety Engineering*, 5(1):14–21, 3 2018. doi:10.1016/j.jsse.2018.01.002.

Wailord: Parsers and Reproducibility for Quantum Chemistry

Rohit Goswami[‡§*]

Abstract—Data driven advances dominate the applied sciences landscape, with quantum chemistry being no exception to the rule. Dataset biases and human error are key bottlenecks in the development of reproducible and generalized insights. At a computational level, we demonstrate how changing the granularity of the abstractions employed in data generation from simulations can aid in reproducible work. In particular, we introduce `wailord` (https://wailord.xyz), a free-and-open-source python library to shorten the gap between data-analysis and computational chemistry, with a focus on the ORCA suite binaries. A two level hierarchy and exhaustive unit-testing ensure the ability to reproducibly describe and analyze "computational experiments". `wailord` offers both input generation, with enhanced analysis, and raw output analysis, for traditionally executed ORCA runs. The design focuses on treating output and input generation in terms of a mini domain specific language instead of more imperative approaches, and we demonstrate how this abstraction facilitates chemical insights.

Index Terms—quantum chemistry, parsers, reproducible reports, computational inference

Introduction

The use of computational methods for chemistry is ubiquitous and few modern chemists retain the initial skepticism of the field [Koh99], [Sch86]. Machine learning has been further earmarked [MSH19], [Dra20], [SGT+19] as an effective accelerator for computational chemistry at every level, from DFT [GLL+16] to alchemical searches [DBCC16] and saddle point searches [ÁJ18]. However, these methods trade technical rigor for vast amounts of data, and so the ability to reproduce results becomes increasingly more important. Independently, the ability to reproduce results [Pen11], [SNTH13] in all fields of computational research, and has spawned a veritable flock of methodological and programmatic advances [CAB+19], including the sophisticated provenance tracking of AiiDA [PCS+16], [HZU+20].

Dataset bias

[EIS+20], [BS19], [RBA+19] has gained prominence in the machine learning literature, but has not yet percolated through to the chemical sciences community. At its core, the argument for dataset biases in generic machine learning problems of image and text classification, can be linked to the difficulty in obtaining labeled results for training purposes. This is not an issue in the computational physical sciences at all, as the training data can often be labeled without human intervention. This is especially true when simulations are carried out at varying levels of accuracy. However, this also leads to a heavy reliance on high accuracy calculations on "benchmark" datasets and results [HMSE+21], [SEJ+19].

Compute is expensive, and the reproduction of data which is openly available is often hard to justify as a valid scientific endeavor. Rather than focus on the observable outputs of calculations, instead we assert that it is best to be able to have reproducible confidence in the elements of the workflow. In the following sections, we will outline `wailord`, a library which implements a two level structure for interacting with ORCA [Nee12] to implement an end-to-end workflow to analyze and prepare datasets. Our focus on ORCA is due to its rapid and responsive development cycles, that it is free to use (but not open source) and also because of its large repertoire of computational chemistry calculations. Notably, the black-box nature of ORCA (in that the source is not available) mirrors that of many other packages (which are not free) like VASP [Haf08]. Using ORCA then, allows us to design a workflow which is best suited for working with many software suites in the community.

We shall understand this `wailord` from the lens of what is often known as a design pattern in the practice of computational science and engineering. That is, a template or description to solve commonly occurring problems in the design of programs.

Structure and Implementation

Python has grown to become the lingua-franca for much of the scientific community [Oli07], [MA11], in no small part because of its interactive nature. In particular, the REPL (read-evaluate-print-loop) structure which has been prioritized (from IPython to Jupyter) is one of the prime motivations for the use of Python as an exploratory tool. Additionally, PyPI, the python package index, accelerates the widespread disambiguation of software packages. Thus `wailord` is implemented as a free and open source `python` library.

Structure

Data generation involves set of known configurations (say, `xyz` inputs) and a series of common calculations whose outputs are required. Computational chemistry packages tend to be focused on acceleration and setup details on a *per-job* scale. `wailord`,

* *Corresponding author: rog32@hi.is*
‡ *Science Institute, University of Iceland*
§ *Quansight Austin, TX, USA*

in contrast, considers the outputs of simulations to form a tree, where the actual run and its inputs are the leaves, and each layer of the tree structure holds information which is collated into a single dataframe which is presented to the user.

Downstream tasks for simulations of chemical systems involve questions phrased as queries or comparative measures. With that in mind, `wailord` generates `pandas` dataframes which are indistinguishable from standard machine learning information sources, to trivialize the data-munging and preparation process. The outputs of `wailord` represent concrete *information* and it is not meant to store runs like the ASE database [LMB+17] , nor run a process to manage discrete workflows like AiiDA [HZU+20].

By construction, it differs also from existing "interchange" formats as those favored by the materials data repositories like the QCArchive project [SAB+21] and is partially close in spirit to the `cclib` endeavor [OTL08].

Implementation

Two classes form the backbone of the data-harvesting process. The intended point of interface with a user is the `orcaExp` class which collects information from multiple ORCA outputs and produces dataframes which include relevant metadata (theory, basis, system, etc.) along with the requested results (energy surfaces, energies, angles, geometries, frequencies, etc.). A lower level "orca visitor" class is meant to parse each individual ORCA output. Until the release of ORCA 5 which promises structured property files, the outputs are necessarily parsed with regular expressions, but validated extensively. The focus on ORCA has allowed for more exotic helper functions, like the calculation of rate constants from `orcaVis` files. However, beyond this functionality offered by the quantum chemistry software (ORCA), a computational chemistry workflow requires data to be more malleable. To this end, the plain-text or binary outputs of quantum chemistry software must be further worked on (post-processed) to gain insights. This means for example, that the outputs may be entered into a spreadsheet, or into a plain text note, or a lab notebook, but in practice, programming languages are a good level of abstraction. Of the programming languages, Python as a general purpose programming language with a high rate of community adoption is a good starting place.

Python has a rich set of structures implemented in the standard library, which have been liberally used for structuring outputs. Furthermore, there have been efforts to convert the grammar of graphics [WW05] and tidy-data [WAB+19] approaches to the `pandas` package which have also been adapted internally, including strict unit adherence using the `pint` library. The user is not burdened by these implementation details and is instead ensured a `pandas` data-frame for all operations, both at the `orcaVis` level, and the `orcaExp` level.

Software industry practices have been followed throughout the development process. In particular, the entire package is written in a test-driven-development (TDD) fashion which has been proven many times over for academia [DJS08] and industry [BN06]. In essence, each feature is accompanied by a test-case. This is meant to ensure that once the end-user is able to run the test-suite, they are guaranteed the features promised by the software. Additionally, this means that potential bugs can be submitted as a test case which helps isolate errors for fixes. Furthermore, software testing allows for coverage metrics, thereby enhancing user and development confidence in different components of any large code-base.

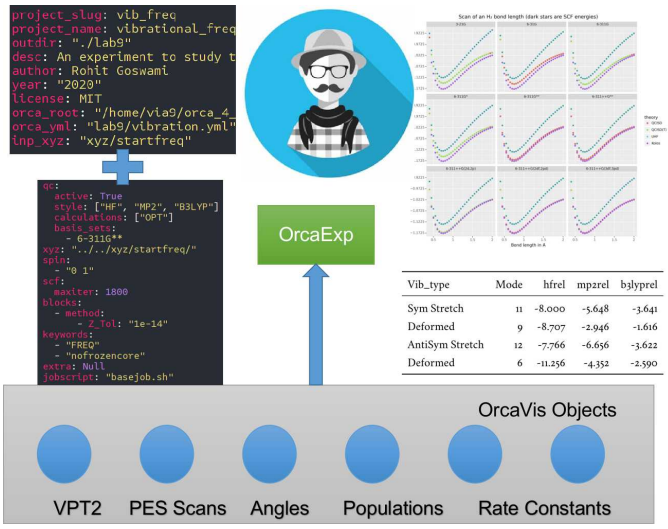


Fig. 1: *Some implemented workflows including the two input YML files. VPT2 stands for second-order vibrational perturbation theory and `Orca_vis` objects are part of `wailord`'s class structure. PES stands for potential energy surface.*

User Interface

The core user interface is depicted in Fig. [[fig:uiwail]]. The test suites cover standard usage and serve as ad-hoc tutorials. Additionally, `jupyter` notebooks are also able to effectively run `wailord` which facilitates its use over SSH connections to high-performance-computing (HPC) clusters. The user is able to describe the nature of calculations required in a simple YAML file format. A command line interface can then be used to generate inputs, or another YAML file may be passed to describe the paths needed. A very basic harness script for submissions is also generated which can be rate limited to ensure optimal runs on an HPC cluster.

Design and Usage

A simulation study can be broken into:

- Inputs + Configuration for runs + Data for structures
- Outputs per run
- Post-processing and aggregation

From a software design perspective, it is important to recognize the right level of abstraction for the given problem. An object-oriented pattern is seen to be the correct design paradigm. However, though combining test driven development and object oriented design is robust and extensible, the design of `wailord` is meant to tackle the problem at the level of a domain specific language. Recall from formal language theory [AA07] the fact that a grammar is essentially meant to specify the entire possible set of inputs and outputs for a given language. A grammar can be expressed as a series of tokens (terminal symbols) and non-terminal (syntactic variables) symbols along with rules defining valid combinations of these.

It may appear that there is little but splitting hairs between parsing data line by line as is traditionally done in libraries, compared to defining the exact structural relations between allowed symbols. However, this design, apart from disallowing invalid inputs, also makes sense from a pedagogical perspective.

For example, of the inputs, structured data like configurations (XYZ formats) are best handled by concrete grammars, where each rule is followed in order:

```
grammar_xyz = Grammar(
    r"""
    meta = natoms ws coord_block ws?
    natoms = number
    coord_block = (aline ws)+
    aline = (atype ws cline)
    atype = ~"[a-zA-Z]" / ~"[0-9]"
    cline = (float ws float ws float)
    float = pm number "." number
    pm              = ~"[+-]?"
    number          = ~"\\d+"
    ws              = ~"\\s*"
    """
)
```

This definition maps neatly into the exact specification of an `xyz` file:

```
2

H   -2.8   2.8   0.1
H   -3.2   3.4   0.2
```

Where we recognize that the overarching structure is of the number of atoms, followed by multiple coordinate blocks followed by optional whitespace. We move on to define each coordinate block as a line of one or many `aline` constructs, each of which is an `atype` with whitespace and three float values representing coordinates. Finally we define the positive, negative, numeric and whitespace symbols to round out the grammar. This is the exact form of every valid `xyz` file. The `parsimonious` library allows handling grammatical constructs in a Pythonic manner.

However, the generation of inputs is facilitated through the use of generalized templates for "experiments" controlled by `cookiecutter`. This allows for validations on the workflow during setup itself.

For the purposes of the simulation study, one "experiment" consists of multiple single-shot runs; each of which can take a long time.

Concretely, the top-level "experiment" is controlled by a YAML file:

```
project_slug: methylene
project_name: singlet_triplet_methylene
outdir: "./lab6"
desc: An experiment to calculate singlet and triplet
states differences at a QCISD(T) level
author: Rohit
year: "2020"
license: MIT
orca_root: "/home/orca/"
orca_yml: "orcaST_meth.yml"
inp_xyz: "ch2_631ppg88_trip.xyz"
```

Where each run is then controlled individually.

```
qc:
  active: True
  style: ["UHF", "QCISD", "QCISD(T)"]
  calculations: ["OPT"]
  basis_sets:
    - 6-311++G**
xyz: "inp.xyz"
spin:
  - "0 1" # Singlet
  - "0 3" # Triplet
extra: "!NUMGRAD"
viz:
  molden: True
  chemcraft: True
jobscript: "basejob.sh"
```

Usage is then facilitated by a high-level call.

```
waex.cookies.gen_base(
template="basicExperiment",
absolute=False,
filen="./lab6/expCookieST_meth.yml",
)
```

The resulting directory tree can be sent to a High Performance Computing Cluster (HPC), and once executed via the generated run-script helper; locally analysis can proceed.

```
mdat = waio.orca.genEBASet(Path("buildOuts") / \
"methylene",
deci=4)
print(mdat.to_latex(index=False,
caption="CH2 energies and angles \
at various levels of theory, with NUMGRAD"))
```

In certain situations, ordering may be relevant as well (e.g. for generating curves of varying density functional theoretic complexity). This can be handled as well.

For the outputs, similar to the key ideas across `signac`, `nix`, `spack` and other tools, control is largely taken away from the user in terms of the auto-generated directory structure. The outputs of each run is largely collected through regular expressions, due to the ever changing nature of the outputs of closed source software.

Importantly, for a code which is meant to confer insights, the concept of units is key. `wailord` with `ORCA` has first class support for units using `pint`.

Dissociation of H2

As a concrete example, we demonstrate a popular pedagogical exercise, namely to obtain the binding energy curves of the H2 molecule at varying basis sets and for the Hartree Fock, along with the results of Kolos and Wolniewicz [KW68]. We first recognize, that even for a moderate 9 basis sets with 33 points, we expect around 1814 data points. Where each basis set requires a separate run, this is easily expected to be tedious.

Naively, this would require modifying and generating ORCA input files.

```
!UHF 3-21G ENERGY

%paras
    R = 0.4, 2.0, 33 # x-axis of H1
end

*xyz 0 1
H    0.00    0.0000000    0.0000000
H    {R}     0.0000000    0.0000000
*
```

We can formulate the requirement imperatively as:

```
qc:
  active: True
  style: ["UHF", "QCISD", "QCISD(T)"]
  calculations: ["ENERGY"] # Same as single point or SP
  basis_sets:
    - 3-21G
    - 6-31G
    - 6-311G
    - 6-311G*
    - 6-311G**
    - 6-311++G**
    - 6-311++G(2d,2p)
    - 6-311++G(2df,2pd)
    - 6-311++G(3df,3pd)
xyz: "inp.xyz"
spin:
  - "0 1"
params:
  - name: R
```

```
    range: [0.4, 2.00]
    points: 33
    slot:
      xyz: True
      atype: "H"
      anum: 1 # Start from 0
      axis: "x"
extra: Null
jobscript: "basejob.sh"
```

This run configuration is coupled with an experiment setup file, similar to the one in the previous section. With this in place, generating a data-set of all the required data is fairly trivial.

```
kolos = pd.read_csv(
    "../kolos_H2.ene",
    skiprows=4,
    header=None,
    names=["bond_length", "Actual Energy"],
    sep=" ",
)
kolos['theory']="Kolos"

expt = waio.orca.orcaExp(expfolder=Path("buildOuts") /
h2dat = expt.get_energy_surface()
```

Finally, the resulting data can be plotted using tidy principles.

```
imgname = "images/plotH2A.png"
p1a = (
    p9.ggplot(
        data=h2dat, mapping=p9.aes(x="bond_length",
        y="Actual Energy",
        color="theory")
    )
    + p9.geom_point()
    + p9.geom_point(mapping=p9.aes(x="bond_length",
      y="SCF Energy"),
      color="black", alpha=0.1,
      shape='*', show_legend=True)
    + p9.geom_point(mapping=p9.aes(x="bond_length",
      y="Actual Energy",
      color="theory"),
      data=kolos,
      show_legend=True)
    + p9.scales.scale_y_continuous(breaks
      = np.arange( h2dat["Actual Energy"].min(),
      h2dat["Actual Energy"].max(), 0.05) )
    + p9.ggtitle("Scan of an H2 \
      bond length (dark stars are SCF energies)")
    + p9.labels.xlab("Bond length in Angstrom")
    + p9.labels.ylab("Actual Energy (Hatree)")
    + p9.facet_wrap("basis")
)
p1a.save(imgname, width=10, height=10, dpi=300)
```

Which gives rise to the concise representation Fig. 2 from which all required inference can be drawn.

In this particular case, it is possible to see the deviations from the experimental results at varying levels of theory for different basis sets.

Conclusions

We have discussed `wailord` in the context of generating, in a reproducible manner the structured inputs and output datasets which facilitate chemical insight. The formulation of bespoke datasets tailored to the study of specific properties across a wide range of materials at varying levels of theory has been shown. The test-driven-development approach is a robust methodology for interacting with closed source software. The design patterns expressed, of which the `wailord` library is a concrete implementation, is expected to be augmented with more workflows, in particular, with a focus on nudged elastic band. The methodology

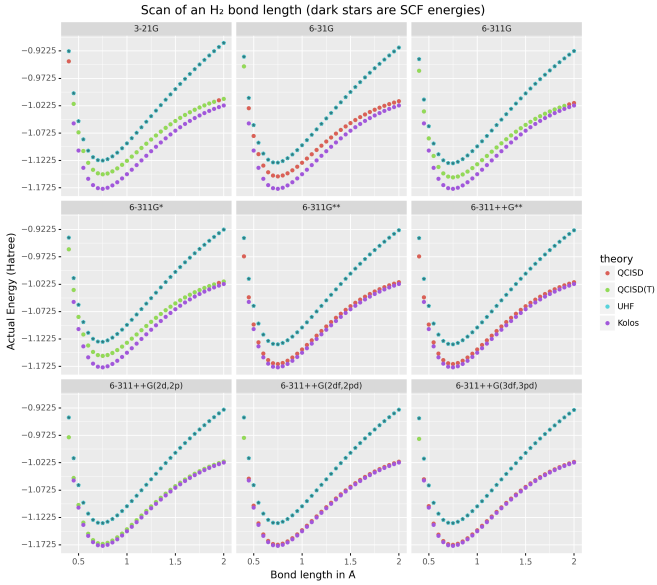


Fig. 2: *Plots generated from tidy principles for post-processing* `wailord` *parsed outputs.*

here has been applied to ORCA, however, the two level structure has generalizations to most quantum chemistry codes as well.

Importantly, we note that the ideas expressed form a design pattern for interacting with a plethora of computational tools in a reproducible manner. By defining appropriate scopes for our structured parsers, generating deterministic directory trees, along with a judicious use of regular expressions for output data harvesting, we are able to leverage tidy-data principles to analyze the results of a large number of single-shot runs.

Taken together, this tool-set and methodology can be used to generate elegant reports combining code and concepts together in a seamless whole. Beyond this, the interpretation of each computational experiment in terms of a concrete domain specific language is expected to reduce the requirement of having to re-run benchmark calculations.

Acknowledgments

R Goswami thanks H. Jónsson and V. Ásgeirsson for discussions on the design of computational experiments for inference in computation chemistry. This work was partially supported by the Icelandic Research Fund, grant number 217436052.

References

[AA07] Alfred V. Aho and Alfred V. Aho, editors. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, Boston, 2nd ed edition, 2007.

[ÁJ18] Vilhjálmur Ásgeirsson and Hannes Jónsson. Exploring Potential Energy Surfaces with Saddle Point Searches. In Wanda Andreoni and Sidney Yip, editors, *Handbook of Materials Modeling*, pages 1–26. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-42913-7_28-1.

[BN06] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 356–363, New York, NY, USA, September 2006. Association for Computing Machinery. doi:10.1145/1159733.1159787.

[BS19] Avrim Blum and Kevin Stangl. Recovering from Biased Data: Can Fairness Constraints Improve Accuracy? *arXiv:1912.01094 [cs, stat]*, December 2019. arXiv:1912.01094.

[CAB+19] The Turing Way Community, Becky Arnold, Louise Bowler, Sarah Gibson, Patricia Herterich, Rosie Higman, Anna Krystalli, Alexander Morley, Martin O'Reilly, and Kirstie Whitaker. The Turing Way: A Handbook for Reproducible Data Science. Zenodo, March 2019.

[DBCC16] Sandip De, Albert P. Bartók, Gábor Csányi, and Michele Ceriotti. Comparing molecules and solids across structural and alchemical space. *Physical Chemistry Chemical Physics*, 18(20):13754–13769, May 2016. doi:10.1039/C6CP00415F.

[DJS08] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, June 2008. doi:10.1145/1383602.1383644.

[Dra20] Pavlo O. Dral. Quantum Chemistry in the Age of Machine Learning. *The Journal of Physical Chemistry Letters*, 11(6):2336–2347, March 2020. doi:10.1021/acs.jpclett.9b03664.

[EIS+20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Jacob Steinhardt, and Aleksander Madry. Identifying Statistical Bias in Dataset Replication. *arXiv:2005.09619 [cs, stat]*, May 2020. arXiv:2005.09619.

[GLL+16] Ting Gao, Hongzhi Li, Wenze Li, Lin Li, Chao Fang, Hui Li, Li-Hong Hu, Yinghua Lu, and Zhong-Min Su. A machine learning correction for DFT non-covalent interactions based on the S22, S66 and X40 benchmark databases. *Journal of Cheminformatics*, 8(1):24, May 2016. doi:10.1186/s13321-016-0133-7.

[Haf08] Jürgen Hafner. Ab-initio simulations of materials using VASP: Density-functional theory and beyond. *Journal of Computational Chemistry*, 29(13):2044–2078, 2008. doi:10.1002/jcc.21057.

[HMSE+21] Johannes Hoja, Leonardo Medrano Sandonas, Brian G. Ernst, Alvaro Vazquez-Mayagoitia, Robert A. DiStasio Jr., and Alexandre Tkatchenko. QM7-X, a comprehensive dataset of quantum-mechanical properties spanning the chemical space of small organic molecules. *Scientific Data*, 8(1):43, February 2021. doi:10.1038/s41597-021-00812-2.

[HZU+20] Sebastiaan P. Huber, Spyros Zoupanos, Martin Uhrin, Leopold Talirz, Leonid Kahle, Rico Häuselmann, Dominik Gresch, Tiziano Müller, Aliaksandr V. Yakutovich, Casper W. Andersen, Francisco F. Ramirez, Carl S. Adorf, Fernando Gargiulo, Snehal Kumbhar, Elsa Passaro, Conrad Johnston, Andrius Merkys, Andrea Cepellotti, Nicolas Mounet, Nicola Marzari, Boris Kozinsky, and Giovanni Pizzi. AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Scientific Data*, 7(1):300, September 2020. doi:10.1038/s41597-020-00638-4.

[Koh99] W. Kohn. Nobel Lecture: Electronic structure of matter—wave functions and density functionals. *Reviews of Modern Physics*, 71(5):1253–1266, October 1999. doi:10.1103/RevModPhys.71.1253.

[KW68] W. Kolos and L. Wolniewicz. Improved Theoretical Ground-State Energy of the Hydrogen Molecule. *The Journal of Chemical Physics*, 49(1):404–410, July 1968. doi:10.1063/1.1669836.

[LMB+17] Ask Hjorth Larsen, Jens Jørgen Mortensen, Jakob Blomqvist, Ivano E. Castelli, Rune Christensen, Marcin Du\lak, Jesper Friis, Michael N. Groves, Bjørk Hammer, Cory Hargus, Eric D. Hermes, Paul C. Jennings, Peter Bjerre Jensen, James Kermode, John R. Kitchin, Esben Leonhard Kolsbjerg, Joseph Kubal, Kristen Kaasbjerg, Steen Lysgaard, Jón Bergmann Maronsson, Tristan Maxson, Thomas Olsen, Lars Pastewka, Andrew Peterson, Carsten Rostgaard, Jakob Schiøtz, Ole Schütt, Mikkel Strange, Kristian S. Thygesen, Tejs Vegge, Lasse Vilhelmsen, Michael Walter, Zhenhua Zeng, and Karsten W. Jacobsen. The atomic simulation environment—a Python library for working with atoms. *Journal of Physics: Condensed Matter*, 29(27):273002, June 2017. doi:10.1088/1361-648X/aa680e.

[MA11] K. J. Millman and M. Aivazis. Python for Scientists and Engineers. *Computing in Science Engineering*, 13(2):9–12, March 2011. doi:10/dc343g.

[MSH19] Ralf Meyer, Klemens S. Schmuck, and Andreas W. Hauser. Machine Learning in Computational Chemistry: An Evaluation of Method Performance for Nudged Elastic Band Calculations. *Journal of Chemical Theory and Computation*, 15(11):6513–6523, November 2019. doi:10.1021/acs.jctc.9b00708.

[Nee12] Frank Neese. The ORCA program system. *WIREs Computational Molecular Science*, 2(1):73–78, 2012. doi:10.1002/wcms.81.

[Oli07] T. E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, May 2007. doi:10/fjzzc8.

[OTL08] Noel M. O'boyle, Adam L. Tenderholt, and Karol M. Langner. Cclib: A library for package-independent computational chemistry algorithms. *Journal of Computational Chemistry*, 29(5):839–845, 2008. doi:10.1002/jcc.20823.

[PCS+16] Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky. AiiDA: Automated interactive infrastructure and database for computational science. *Computational Materials Science*, 111:218–230, January 2016. doi:10.1016/j.commatsci.2015.09.013.

[Pen11] Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, December 2011. doi:10/fdv356.

[RBA+19] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the Spectral Bias of Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 5301–5310. PMLR, May 2019.

[SAB+21] Daniel G. A. Smith, Doaa Altarawy, Lori A. Burns, Matthew Welborn, Levi N. Naden, Logan Ward, Sam Ellis, Benjamin P. Pritchard, and T. Daniel Crawford. The MolSSI QCArchive project: An open-source platform to compute, organize, and share quantum chemistry data. *WIREs Computational Molecular Science*, 11(2):e1491, 2021. doi:10.1002/wcms.1491.

[Sch86] Henry F. Schaefer. Methylene: A Paradigm for Computational Quantum Chemistry. *Science*, 231(4742):1100–1107, March 1986. doi:10.1126/science.231.4742.1100.

[SEJ+19] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Protein structure prediction using multiple deep neural networks in the 13th Critical Assessment of Protein Structure Prediction (CASP13). *Proteins: Structure, Function, and Bioinformatics*, 87(12):1141–1148, 2019. doi:10.1002/prot.25834.

[SGT+19] K. T. Schütt, M. Gastegger, A. Tkatchenko, K.-R. Müller, and R. J. Maurer. Unifying machine learning and quantum chemistry with a deep neural network for molecular wavefunctions. *Nature Communications*, 10(1):5024, November 2019. doi:10.1038/s41467-019-12875-2.

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

[WAB+19] Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the Tidyverse. *Journal of Open Source Software*, 4(43):1686, November 2019. doi:10.21105/joss.01686.

[WW05] Leland Wilkinson and Graham Wills. *The Grammar of Graphics*. Statistics and Computing. Springer, New York, 2nd ed edition, 2005.

Variational Autoencoders For Semi-Supervised Deep Metric Learning

Nathan Safir[‡*], Meekail Zain[§], Curtis Godwin[‡], Eric Miller[‡], Bella Humphrey[§], Shannon P Quinn[§¶]

Abstract—Deep metric learning (DML) methods generally do not incorporate unlabelled data. We propose borrowing components of the variational autoencoder (VAE) methodology to extend DML methods to train on semi-supervised datasets. We experimentally evaluate the atomic benefits to the perform- ing DML on the VAE latent space such as the enhanced ability to train using unlabelled data and to induce bias given prior knowledge. We find that jointly training DML with an autoencoder and VAE may be potentially helpful for some semi-suprevised datasets, but that a training routine of alternating between the DML loss and an additional unsupervised loss across epochs is generally unviable.

Index Terms—Variational Autoencoders, Metric Learning, Deep Learning, Representation Learning, Generative Models

Introduction

Within the broader field of representation learning, metric learning is an area which looks to define a distance metric which is smaller between similar objects (such as objects of the same class) and larger between dissimilar objects. Oftentimes, a map is learned from inputs into a low-dimensional latent space where euclidean distance exhibits this relationship, encouraged by training said map against a loss (cost) function based on the euclidean distance between sets of similar and dissimilar objects in the latent space. Existing metric learning methods are generally unable to learn from unlabelled data, which is problematic because unlabelled data is often easier to obtain and is potentially informative.

We take inspiration from variational autoencoders (VAEs), a generative representation learning architecture, for using unlabelled data to create accurate representations. Specifically, we look to evaluate three atomic improvement proposals that detail how pieces of the VAE architecture can create a better deep metric learning (DML) model on a semi-supervised dataset. From here, we can ascertain which specific qualities of how VAEs process unlabelled data are most helpful in modifying DML methods to train with semi-supervised datasets.

First, we propose that the autoencoder structure of the VAE helps the clustering of unlabelled points, as the reconstruction loss may help incorporate semantic information from unlabelled sources. Second, we propose that the structure of the VAE latent space, as it is confined by a prior distribution, can be used to induce bias in the latent space of a DML system. For instance, if we know a dataset contains N -many classes, creating a prior distribution that is a learnable mixture of N gaussians may help produce better representations. Third, we propose that performing DML on the latent space of the VAE so that the DML task can be jointly optimized with the VAE to incorporate unlabelled data may help produce better representations.

Each of the three improvement proposals will be evaluated experimentally. The improvement proposals will be evaluated by comparing a standard DML implementation to the same DML implementation:

- jointly optimized with an autoencoder
- while structuring the latent space around a prior distribution using the VAE's KL-divergence loss term between the approximated posterior and prior
- jointly optimized with a VAE

Our primary contribution is evaluating these three improvement proposals. Our secondary contribution is presenting the results of the joint approaches for VAEs and DML for more recent metric losses that have not been jointly optimized with a VAE in previous literature.

Related Literature

The goal of this research is to investigate how components of the variational autoencoder can help the performance of deep metric learning in semi supervised tasks. We draw on previous literature to find not only prior attempts at this specific research goal but also work in adjacent research questions that proves insightful. In this review of the literature, we discuss previous related work in the areas of Semi-Supervised Metric Learning and VAEs with Metric Losses.

Semi-Supervised Metric Learning

There have been previous approaches to designing metric learning architectures which incorporate unlabelled data into the metric learning training regimen for semi-supervised datasets. One of the original approaches is the MPCK-MEANS algorithm proposed by Bilenko et al. ([BBM04]), which adds a penalty for placing labelled inputs in the same cluster which are of a different class or in different clusters if they are of the same class. This penalty is proportional to the metric distance between the pair of inputs.

* *Corresponding author: nssafir@gmail.com*
‡ *Institute for Artificial Intelligence, University of Georgia, Athens, GA 30602 USA*
§ *Department of Computer Science, University of Georgia, Athens, GA 30602 USA*
¶ *Department of Cellular Biology, University of Georgia, Athens, GA 30602 USA*

Baghshah and Shouraki ([BS09]) also looks to impose similar constraints by introducing a loss term to preserve locally linear relationships between labelled and unlabelled data in the input space. Wang et al. ([WYF13]) also use a regularizer term to preserve the topology of the input space. Using VAEs, in a sense, draws on this theme: though there is not explicit term to enforce that the topology of the input space is preserved, a topology of the inputs is intended to be learned through a low-dimensional manifold in the latent space.

One more recent common general approach to this problem is to use the unlabelled data's proximity to the labelled data to estimate labels for unlabelled data, effectively transforming unlabelled data into labelled data. Dutta et al. ([DHS21]) and Li et al. ([LYZ+19]) propose a model which uses affinity propagation on a k-Nearest-Neighbors graph to label partitions of unlabelled data based on their closest neighbors in the latent space. Wu et al. ([WFZ20]) also look to assign pseudo-labels to unlabelled data, but not through a graph-based approach. Instead, the proposed model looks to approximate "soft" pseudo-labels for unlabelled data from the metric learning similarity measure between the embedding of unlabelled data and the center of each input of each class of the labelled data.

Several of the recent graph based approaches can be considered state-of-the-art for semi supervised metric learning. Li et. al.'s paper states their methods achieve 98.9 percent clustering accuracy on the MNIST dataset with 10% labelled data, outperforming two similar state-of-the-art methods, DFCM ([ARJM18]) and SDEC ([RHD+19]), by roughly 8 points. Dutta et. al.'s method also outperforms 5 other state for the R@1 metric (the "percentage of test examples" that have at least one 1 "nearest neighbor from the same class.") by at leat 1.2 on the MNIST dataset, as well as the Fashion-MNIST and CIFAR-10 datasets. It is difficult to compare the two approaches as the evaluation metrics used in each paper differ. Li et al.'s paper has been cited rather heavily relative to other papers in the field and can be considered state of the art for semi-supervised DML on MNIST. The paper also provides a helpful metric (98.9 percent clustering accuracy on the MNIST dataset with 10% labelled data) to use as a reference point for the results in this paper.

VAEs with Metric Loss

Some approaches to incorporating labelled data into VAEs use a metric loss to govern the latent space more explicitly. Lin et al. ([LDD+18]) model the intra-class invariance (i.e. the class-related information of a data point) and intra-class variance (i.e. the distinct features of a data point not unique to it's class) seperately. Like several other models in this section, this paper's proposed model incorporates a metric loss term for the latent vectors representing intra-class invariance and the latent vectors representing both intra-class invariance and intra-class variance.

Kulkarni et al. ([KCJ20]) incorporate labelled information into the VAE methodology in two ways. First, a modified architecture called the CVAE is used in which the encoder and generator of the VAE is not only conditioned on the input X and latent vector z, respectively, but also on the label Y. The CVAE was introduced in previous papers ([SLY15]) ([DCGO19]). Second, the authors add a metric loss, specifically a multi-class N-pair loss ([Soh16]), in the overall loss function of the model. While it is unclear how the CVAE technique would be adapted in a semi-supervised setting, as there is not a label Y associated with each datapoint X, we also experiment with adding a (different) metric loss to the overall VAE loss function.

Most recently, Grosnit et al. ([GTM+21]) leverage a new training algorithm for combining VAEs and DML for Bayesian Optimization and said algorithm using simple, contrastive, and triplet metric losses. We look to build on this literature by also testing a combined VAE DML architecture on more recent metric losses, albeit using a simpler training regimen.

Deep Metric Learning (DML)

Metric learning attempts to create representations for data by training against the similarity or dissimilarity of samples. In a more technical sense, there are two notable functions in DML systems. Function f_θ is a neural network which maps the input data X to the latent points Z (i.e. $f_\theta : X \mapsto Z$, where θ is the network parameters). Generally, Z exists in a space of much lower dimensionality than X (eg. X is a set of 28×28 pixel pictures such that $X \subset \mathbb{R}^{28\times 28}$ and $Z \subset \mathbb{R}^{10}$).

The function $D_{f_\theta}(x,y) = D(f_\theta(x), f_\theta(y))$ represents the distance between two inputs $x, y \in X$. To create a useful embedding model f_θ, we would like for f_θ to produce large values of $D_{f_\theta}(x,y)$ when x and y are dissimilar and for f_θ to produce small values of $D_{f_\theta}(x,y)$ when x and y are similar. In some cases, dissimilarity and similarity can refer to when inputs are of different and the same classes, respectively.

It is common for the Euclidean metric (i.e. the L_2 metric) to be used as a distance function in metric learning. The generalized L_p metric can be defined as follows, where $z_0, z1 \in \mathbb{R}^d$.

$$D_p(z_0, z_1) = ||z_0 - z_1||_p = (\sum_{i=1}^{d} |z_{0_i} - z_{1_i}|^p)^{1/p}$$

If we have chosen f_θ (a neural network) and the distance function D (the L_2 metric), the remaining component to be defined in a metric learning system is the loss function for training f. In practice, we will be using triplet loss ([SKP15]), one of the most common metric learning loss functions.

Methodology

We look to discover the potential of applying components of the VAE methodology to DML systems. We test this through presenting incremental modifications to the basic DML architecture. Each modified architecture corresponds to an improvement proposal about how a specific part of the VAE training regime and loss function may be adapted to assist the performance of a DML method for a semi-supervised dataset.

The general method we will take for creating modified DML models involves extending the training regimen to two phases, a supervised and unsupervised phase. In the supervised phase the modified DML model behaves identically to the base DML model, training on the same metric loss function. In the unsupervised phase, the DML model will train against an unsupervised loss inspired by the VAE. This may require extra steps to be added to the DML architecture. In the pseudocode, s refers to boolean variable representing if the current phase is supervised. α is a hyperparameter which modulates the impact of the unsupervised on total loss for the DML autoencoder.

Algorithm 1: Base DML Training Routine

Input: Dataset dataset D, encoder network f, metric loss function m, learning rate γ, weights θ
Result: updated weights θ
for *batch* x, y *in* D **do**
  $z = f_\theta(x)$;
  $c = m(z, y)$;
  Compute $\frac{dc}{d\theta}$ with backpropogation;
  $\theta = \theta - \gamma\frac{dc}{d\theta}$;
end

Improvement Proposal 1

We first look to evaluate the improvement proposal that adding a reconstruction loss to a DML system can improve the quality of clustering in the latent representations on a semi-supervised dataset. Reconstruction loss in and of itself enforces a similar semantic mapping onto the latent space as a metric loss, but can be computed without labelled data. In theory, we believe that the added constraint that the latent vector must be reconstructed to approximate the original output will train the spatial positioning to reflect semantic information. Following this reasoning, observations which share similar semantic information, specifically observations of the same class (even if not labelled as such), should intuitively be positioned nearby within the latent space. To test if this intuition occurs in practice, we evaluate if a DML model with an autoencoder structure and reconstruction loss (described in further detail below) will perform better than a plain DML model in terms of clustering quality. This will be especially evident for semi-supervised datasets in which the amount of labelled data is not feasible for solely supervised DML.

Given a semi-supervised dataset, we assume a standard DML system will use only the labelled data and train given a metric loss L_{metric} (see Algorithm 1). Our modified model DML Autoencoder will extend the DML model's training regime by adding a decoder network which takes the latent point z as input and produces an output $\hat{x}$. The unsupervised loss L_U is equal to the reconstruction loss.

Improvement Proposal 2

Say we are aware that a dataset has n classes. It may be useful to encourage that there are n clusters in the latent space of a DML model. This can be enforced by using a prior distribution containing n many Gaussians. As we wish to measure only the affect of inducing bias on the representation without adding any complexity to the model, the prior distribution will not be learnable (unlike VAE with VampPrior). By testing whether the classes of points in the latent space are organized along the prior components we can test whether bias can be induced using a prior to constrain the latent space of a DML. By testing whether clustering improves performance, we can evaluate whether this inductive bias is helpful.

Given a fully supervised dataset, we assume a standard DML system will use only the labelled data and train given a metric loss L_{metric}. Our modified model will extend the DML system's training regime by setting the unsupervised loss to a KL divergence term that measures the difference between posterior distributions and a prior distribution. It should also be noted that, like the VAE encoder, we will map the input not to a latent point but to a latent distribution. The latent point is stochastically sampled from the latent distribution during training. Mapping the input to a distribution instead of a point will allow us to calculate the KL divergence.

In practice, we will be evaluating a DML model with a unit prior and a DML model with a mixture of gaussians (GMM) prior. The latter model constructs the prior as a mixture of n gaussians – each the vertice of the unit (i.e. each side is 2 units long) hypercube in the latent space. The logvar of each component is set equal to one. Constructing the prior in this way is beneficial in that it is ensured that each component is evenly spaced within the latent space, but is limiting in that there must be exactly 2^d components in the GMM prior. Thus, to test, we will test a dataset with 10 classes on the latent space dimensionality of 4, such that there are $2^4 = 16$ gaussian components in the GMM prior. Though the number of prior components is greater than the number of classes, the latent mapping may still exhibit the pattern of classes forming clusters around the prior components as the extra components may be made redundant.

The drawback of the decision to set the GMM components' means to the coordinates of the unit hypercube's vertices is that the manifold of the chosen dataset may not necessarily exist in 4 dimensions. Choosing gaussian components from a d-dimensional hypersphere in the latent space $\mathscr{R}^d$ would solve this issue, but there does not appear to be a solution for choosing n evenly spaced points spanning d dimensions on a d-dimensional hypersphere. KL Divergence is calculated with a monte carlo approximation for the GMM and analytically with the unit prior.

Improvement Proposal 3

The third improvement proposal we look to evaluate is that given a semi-supervised dataset, optimizing a DML model jointly with a VAE on the VAE's latent space will produce superior clustering than the DML model individually. The intuition behind this approach is that DML methods can learn from only supervised data and VAE methods can learn from only unsupervised data; the proposed methodology will optimize both tasks simultaneously to learn from both supervised and unsupervised data.

The MetricVAE implementation we create jointly optimizes the VAE task and DML task on the VAE latent space. The unsupervised loss is set to the VAE loss. The implementation uses the VAE with VampPrior model instead of the vanilla VAE.

Results

Experimental Configuration

Each set of experiments shares a similar hyperparameter search space. Below we describe the hyperparameters that are included in the search space of each experiment and the evaluation method.

Learning Rate (lr): Through informal experimentation, we have found that the learning rate of 0.001 causes the models to converge consistently (relative to 0.005 and 0.0005). The learning rate is thus set to 0.001 in each experiment.

Algorithm 2: DML Autoencoder Training Routine

Input: Dataset dataset D, encoder network f, decoder network g, metric loss function m, learning rate γ, coefficient α, weights θ
Result: updated weight θ
for *batch* x, y *in* D **do**
  $z = f_\theta(x)$;
  $\hat{x} = g_\theta(z)$;
  if s **then**
    $c = (1-\alpha) * m(z, y) + \alpha * MSE(\hat{x}, x)$;
  else
    $c = \alpha * MSE(\hat{x}, x)$;
  end
  Compute $\frac{dc}{d\theta}$ with backpropogation;
  $\theta = \theta - \gamma \frac{dc}{d\theta}$;
end

Algorithm 3: DML with Prior Training Routine

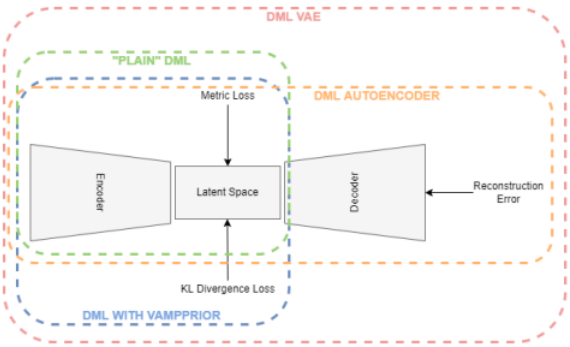
Input: Dataset dataset D, encoder network f, metric loss function m, learning rate γ, coefficient α, prior distribution mean and log-variance μ_p, σ_p, weights θ
Result: updated weights θ
for *batch* x, y *in* D **do**
  $\mu, \sigma = f_\theta(x)$;
  $z \sim N(\mu, \sigma)$;
  if s **then**
    $c = (1-\alpha) * m(z, y) + \alpha * KL(z, \mu, \sigma, \mu_p, \sigma_p)$;
  else
    $c = \alpha * KL(z, \mu, \sigma, \mu_p, \sigma_p)$;
  end
  Compute $\frac{dc}{d\theta}$ with backpropogation;
  $\theta = \theta - \gamma \frac{dc}{d\theta}$;
end

Algorithm 4: Monte-Carlo KL Divergence Algorithm

Input: Latent variable z, approximate posterior distribution mean and variance μ,σ, prior distribution mean and log-variance μ_p, σ_p
Result: KL Divergence between distributions q and p
$P(z|\mu, \sigma) = -0.5 * \frac{(z-\mu)^2}{\exp \sigma}$;
$Q(z|\mu_p, \sigma_p) = -0.5 * \frac{(z-\mu_p)^2}{\exp \sigma_p}$;
return $Q(z|\mu_p, \sigma_p) - P(z|\mu, \sigma)$

Algorithm 5: DML VAE Training Routine

Input: Dataset dataset D, encoder network f, decoder network g, metric loss function m, learning rate γ, coefficient α, coefficient β, prior distribution mean and log-variance μ_p, σ_p, weights θ
Result: updated weights θ
for *batch* x, y *in* D **do**
  $\mu, \sigma = f_\theta(x)$;
  $z \sim N(\mu, \sigma)$;
  $\hat{x} = g_\theta(z)$;
  $c_{vae} = MSE(x, x_{hat}) + \beta * KL(z, \mu, \sigma, \mu_p)$;
  if s **then**
    $c = (1-\alpha) * m(z, y) + \alpha * c_{vae}$;
  else
    $c = \alpha * c_{vae}$;
  end
  Compute $\frac{dc}{d\theta}$ with backpropogation;
  $\theta = \theta - \gamma \frac{dc}{d\theta}$;
end



Latent Space Dimensionality (lsdim): Latent space dimensionality refers to the dimensionality of the vector output of the encoder of a DML network or the dimensionality of the posterior distribution of a VAE (also the dimensionality of the latent space). When the latent space dimensionality is 2, we see the added benefit of creating plots of the latent representations (though we can accomplish this through using dimensionality reduction methods like tSNE for higher dimensionalities as well). Example values for this hyperparameter used in experiments are 2, 4, and 10.

Alpha: Alpha (α) is a hyperapameter which refers to the balance between the unsupervised and supervised losses of some of the modified DML models. More details about the role of α in the model implementations are discussed in the methodology section of the model. Potential values for alpha are each between 0 (exclusive) and 1 (inclusive). We do not include 0 in this set as if α is set to 0, the model is equivalent to the fully supervised plain DML model because the supervised loss would not be included. If α is set to 1, then the model would train on only the unsupervised loss; for instance if the DML Autoencoder had α set to 1, then the model would be equivalent to an autoencoder.

Partial Labels Percentage (pl%): The partial labels percentage hyperparameter refers to the percentage of the dataset that is labelled and thus the size of the partion of the dataset that can be used for labelled training. Of course, each of the datasets we use is fully labelled, so a partially labelled datset can be trivially constructed by ignoring some of the labels. As the sizes of the dataset vary, each percentage can refer to a different number of labelled samples. Values for the partial label percentage we use across experiments include 0.01, 0.1, and 10 (with each value referring to the percentage).

Datasets: Two datasets are used for evaluating the models. The first dataset is MNIST ([LC10]), a very popular dataset in machine learning containing greyscale images of handwritten digits. The second dataset we use is the organ OrganAMNIST dataset from MedMNIST v2 ([YSW+21]). This dataset contains 2D slices from computed tomography images from the Liver Tumor Segmentation Benchmark – the labels correspond to the classification of 11 different body organs. The decision to use a second dataset was motivated because as the improvement proposals are tested over more datasets, the results supporting the improvement proposals become more generalizable. The decision to use the OrganAMNIST dataset specifically is motivated in part due to the Quinn Research Group working on similar tasks for biomedical imaging ([ZRS+20]). It is also motivated in part because OrganAMNIST is a more difficult dataset, at least for the classfication task, as the leading accuracy for MNIST is .9991 ([ALP+20]) while the leading accuracy for OrganAMNIST is .951 ([YSW+21]). The MNIST and OrganAMNIST datasets are similar in dimensionality (1 x 28 x 28), number of samples (60,000 and 58,850, respectively) and in that they are both greyscale.

Evaluation: We evaluate the results by running each model on a test partition of data. We then take the latent points Z generated by the model and the corresponding labels Y. Three classifiers (sklearn's implementation of RandomForest, MLP, and kNN) each output predicted labels $\hat{Y}$ for the latent points. In most of the charts shown, however, we only include the kNN classification output due to space constraints and the lack of meaningful difference between the output for each classifier. We finally measure the quality of the predicted labels $\hat{Y}$ using the Adjusted Mutual Information Score (AMI) ([**?**]) and accuracy (which is still helpful but is also easier to interpret in some cases). This scoring metric is common in research that looks to evaluate clustering performance ([ZG21]) ([EKGB16]). We will be using sklearn's implementation of AMI ([PVG+11]). The performance of a classifier on the latent points intuitively can be used as a measure of quality of clustering.

Improvement Proposal 1 Results: Benefits of Reconstruction Loss

In evaluating the first improvement proposal, we compare the performance of the plain DML model to the DML Autoencoder model. We do so by comparing the performance of the plain DML system and the DML Autoencoder across a search space containing the lsdim, alpha, and pl% hyperparameters and both datasets.

In Table 1 and Table 2, we observe that for relatively small amounts of labelled samples (the partial labels percentages of 0.01 and 0.1 correspond to 6 and 60 labelled samples respectively), the DML Autoencoder severely outperforms the DML model. However, when the number of labelled samples increases (the partial labels percentage of 10 correspond to 6000 labelled samples respectively), the DML model significantly outperforms the DML Autoencoder. This trend is not too surprising, as when there is sufficient data to train unsupervised methods and insufficient data to train supervised method, as is the case for the 0.01 and 0.1 partial label percentages, the unsupervised method will likely perform better.

The data looks to show that adding a reconstruction loss to a DML system can improve the quality of clustering in the latent representations on a semi-supervised dataset when there are small amounts (roughly less than 100 samples) of labelled data and a sufficient quantity of unlabelled data. But an important caveat is that it is not convincing that the DML Autoencoder effectively combined the unsupervised and supervised losses to create a superior model, as a plain autoencoder (i.e. the DML Autoencoder

Fig. 1: *Sample images from the MNIST (left) and OrganAMNIST of MedMNIST (right) datasets*

with $\alpha = 1$) outperforms the DML for the partial labels percentage of or less than 0.1% and underperforms the DML for the partial labels percentage of 10%.

Improvement Proposal 2 Results: Incorporating Inductive Bias with a Prior

In evaluating the second improvement proposal, we compare the performance of the plain DML model to the DML with a unit prior and a DML with a GMM prior. The DML prior with the GMM prior will have 2^2 = 4 gaussian components when lsdim = 2 and 2^4 = 16 components when lsdim = 4. Our broad intention is to see if changing the shape (specifically the number of components) of the prior can induce bias by affecting the pattern of embeddings. We hypothesize that when the GMM prior contains n components and n is slightly greater than or equal to the number of classes, each class will cluster around one of the prior components. We will test this for the GMM prior with 16 components (lsdim = 4) as both the MNIST and MedMNIST datasets have 10 classes. We are unable to set the number of GMM components to 10 as our GMM sampling method only allows for the number of components to equal a power of 2. Bseline models include a plain DML and a DML with a unit prior (the distribution N(0, 1)).

In Table 3, it is very evident that across both datasets, the DML models with any prior distribution all devolve to the null model (i.e. the classifier is no better than random selection). From the visualizations of the latent embeddings, we see that the embedded data for the DML models with priors appears completely random. In the case of the GMM prior, it also does not appear to take on the shape of the prior or reflect the number of components in the prior. This may be due to the training routine of the DML models. As the KL divergence loss, which can be said to "fit" the embeddings to the prior, trains on alternating epochs with the supervised DML loss, it is possible that the two losses are not balanced correctly during the training process. From the discussed results, it is fair to state that adding a prior distribution to a DML model through training the model on the KL divergence between the prior and approximated posterior distributions on alternating epochs does is not an effective way to induce bias in the latent space.

Improvement Proposal 3 Results: Jointly Optimizing DML with VAE

To evaluate the third improvement proposal, we compare the performance of DMLs to MetricVAEs (defined in the previous chapter) across several metric losses. We run experiments for triplet loss, supervised loss, and center loss DML and MetricVAE models. To evaluate the improvement proposal, we will assess whether the model performance improves for the MetricVAE over the DML for the same metric loss and other hyper parameters.

Like the previous improvement proposal, the proposed MetricVAE model does not perform better than the null model. As with improvement proposal 2, it is possible this is because the training routine of alternating between supervised loss (in this case, metric loss) and unsupervised (in this case, VAE loss) is not optimal for training the model.

We have trained a seperate combined VAE and DML model which trains on both the unsupervised and supervised loss each epoch instead of alternating between the two each epoch. In the results for this model, we see that an alpha value of over zero (i.e. incorporating both the supervised metric loss into the overall MVAE loss function) can help improve performance especially among lower dimensionalities. Given our analysis of the data, we see that incorporating the DML loss to the VAE is potentially helpful, but only when training the unsupervised and supervised losses jointly. Even in that case, it is unclear whether the MVAE performs better than the corresponding DML model even if it does perform better than the corresponding VAE model.

Conclusion

Conclusion

In this work, we have set out to determine how DML can be extended for semi-supervised datasets by borrowing components of the variational autoencoder. We have formalized this approach through defining three specific improvement proposals. To evaluate each improvement proposal, we have created several variations of the DML model, such as the DML Autoencoder, DML with Unit/GMM Prior, and MVAE. We then tested the performance of the models across several semi-supervised partitions of two datasets, along with other configurations of hyperparameters.

We have determined from the analysis of our results, there is too much dissenting data to clearly accept any three of the improvement proposals. For improvement proposal 1, while the DML Autoencoder outperforms the DML for semisupervised datasets with small amounts of labelled data, it's peformance is not consistently much better than that of a plain autoencoder which uses no labelled data. For improvement proposal 2, each of the DML models with an added prior performed extremely poorly, near or at the level of the null model. For improvement proposal 3, we see the same extremely poor performance from the MVAE models.

From the results in improvement proposals 1 and 3, we find that there may be potential in incorporating the autoencoder and VAE loss terms into DML systems. However, we were unable to show that any of these improvement proposals would consistently outperform the both the DML and fully unsupervised architectures in semisupervised settings. We also found that the training routine used for the improvement proposals, in which the loss function would alternate between supervised and unsupervised losses each epoch, was not effective. This is especially evident in comparing the two combined VAE DML models for improvement proposal 3.

pl%	lsdim	knn acc	knn MI
0.01	2	**0.2618**	**0.1124**
	10	0.2610	0.1073
0.1	2	0.4256	0.2904
	10	**0.6556**	**0.4983**
10	2	0.7476	0.6200
	10	**0.9502**	**0.8838**

pl%	alpha	lsdim	knn acc	knn MI
0.01	0.25	2	0.3957	0.2815
		10	**0.8351**	**0.7172**
	0.5	2	0.3358	0.1932
		10	0.6688	0.5218
	0.75	2	0.3112	0.1835
		10	0.8323	0.7102
	1	2	0.4166	0.2923
		10	0.8319	0.7089
0.1	0.25	2	0.4349	0.3339
		10	0.8718	0.7616
	0.5	2	0.4033	0.3150
		10	**0.8798**	**0.7725**
	0.75	2	0.3812	0.2771
		10	0.7820	0.6399
	1	2	0.4202	0.3022
		10	0.8589	0.7463
10	0.25	2	0.1028	0
		10	0.8478	0.7258
	0.5	2	0.3465	0.2189
		10	0.7925	0.6606
	0.75	2	0.3536	0.2175
		10	0.8497	0.7373
	1	2	0.5137	0.3793
		10	**0.8570**	**0.7407**

Fig. 2: *Table 1: Comparison of the DML (left) and DML Autoencoder (right) models for the MNIST dataset. Bolded values indicate best performance for each partial labels percentage partition (pl%).*

pl%	lsdim	knn acc	knn MI
0.01	2	0.3844	0.2637
	10	**0.6997**	**0.5529**
0.1	2	0.4323	0.3165
	10	**0.8309**	**0.7129**
10	2	0.4616	0.3614
	10	**0.8487**	**0.7339**

pl%	alpha	lsdim	knn acc	knn MI
0.01	0.25	2	0.3683	0.2685
		10	0.8060	0.6792
	0.5	2	0.3388	0.2146
		10	**0.8378**	**0.7207**
	0.75	2	0.3419	0.2183
		10	0.7568	0.5987
	1	2	0.3844	0.2637
		10	0.6997	0.5529
0.1	0.25	2	0.3717	0.2496
		10	0.8514	0.7381
	0.5	2	0.4314	0.2918
		10	0.8553	0.7419
	0.75	2	0.3896	0.2502
		10	**0.8597**	**0.7454**
	1	2	0.4323	0.3165
		10	0.8309	0.7129
10	0.25	2	0.4316	0.3354
		10	0.8560	0.7394
	0.5	2	0.3824	0.2404
		10	**0.8692**	**0.7589**
	0.75	2	0.5118	0.3891
		10	0.8250	0.7078
	1	2	0.4616	0.3614
		10	0.8487	0.7339

Fig. 3: *Table 2: Comparison of the DML (left) and DML Autoencoder (right) models for the MEDMNIST dataset..*

Future Work

In the future, it would be worthwhile to evaluate these improvement proposals using a different training routine. We have stated previously that perhaps the extremely poor performance of the DML with a prior and MVAE models may be due to alternating on training against a supervised and unsupervised loss. Further research could look to develop or compare several different training routines. One alternative would be alternating between losses at each batch instead of each epoch. Another alternative, specifically for the MVAE, may be first training DML on labelled data, training a GMM on it's outputs, and then using the GMM as the prior distribution for the VAE.

Another potentially interesting avenue for future study is in investigating a fourth improvement proposal: the ability to define a Riemannian metric on the latent space. Previous research has shown a Riemannian metric can be computed on the latent space of the VAE by computing the pull-back metric of the VAE's decoder function ([AHS20]). Through the Riemannian metric we could calculate metric losses such as triplet loss with a geodesic instead of euclidean distance. The geodesic distance may be a more accurate representation of similarity in the latent space than euclidean distance as it accounts for the structure of the input data.

REFERENCES

[AHS20] Georgios Arvanitidis, Søren Hauberg, and Bernhard Schölkopf. Geometrically enriched latent spaces. *arXiv preprint arXiv:2008.00565*, 2020. doi:10.48550/arXiv.2008.00565.

[ALP+20] Sanghyeon An, Min Jun Lee, Sanglee Park, Heerin Yang, and Jungmin So. An ensemble of simple convolutional neural network models for MNIST digit recognition. *CoRR*, abs/2008.10400, 2020. URL: https://arxiv.org/abs/2008.10400, arXiv:2008.10400, doi:10.48550/arXiv.2008.10400.

[ARJM18] Ali Arshad, Saman Riaz, Licheng Jiao, and Aparna Murthy. Semi-supervised deep fuzzy c-mean clustering for software fault prediction. *IEEE Access*, 6:25675–25685, 2018. doi:10.1109/ACCESS.2018.2835304.

[BBM04] Mikhail Bilenko, Sugato Basu, and Raymond J Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 11, 2004. doi:10.1145/1015330.1015360.

[BS09] Mahdieh Soleymani Baghshah and Saeed Bagheri Shouraki. Semi-supervised metric learning using pairwise constraints. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[DCGO19] Sara Dahmani, Vincent Colotte, Valérian Girard, and Slim Ouni. Conditional variational auto-encoder for text-driven expressive audiovisual speech synthesis. In *INTERSPEECH 2019-20th Annual Conference of the International Speech Communication Association*, 2019. doi:10.21437/interspeech.2019-2848.

pl%	lsdim	knn acc	knn MI
0.01	2	0.3941	0.2651
	4	0.3791	0.2262
0.10	2	0.4286	0.3135
	4	0.2723	0.1519
10	2	0.4625	0.3653
	4	0.9319	0.8490

pl%	alpha	lsdim	knn acc	knn MI
0.01	0.33	2	0.0962	-0.0008
		4	0.0905	0.0005
	0.66	2	0.1007	0
		4	0.0998	0
	1	2	0.1061	-0.0006
		4	0.1088	0.0013
0.10	0.33	2	0.1064	0.0002
		4	0.1031	0
	0.66	2	0.1061	-0.0011
		4	0.1016	-0.0006
	1	2	0.0959	-0.0005
		4	0.1058	-0.0005
10	0.33	2	0.0950	-0.003
		4	0.1034	0
	0.66	2	0.1043	0
		4	0.1088	0.0007
	1	2	0.0995	-0.0013
		4	0.1055	0.0010

knn acc	knn MI
0.1016	-0.0009
0.0983	0.0005
0.1043	-0.0006
0.1004	-0.0002
0.0935	0.0017
0.1079	-0.0002
0.1079	-0.0005
0.1019	-0.0001
0.1085	0.0012
0.1049	-0.0009
0.0971	0.0007
0.0974	0.0006
0.1052	0.0005
0.0971	-0.0004
0.0965	0.0009
0.09628	-0.0001
0.1112	0
0.1073	0.0012

Fig. 4: *Table 3: Comparison of the DML model (left) and the DML with prior models with a unit gaussian prior (center) and GMM prior (right) models for the MNIST dataset.*

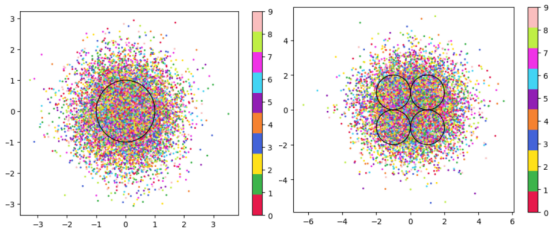


Fig. 5: *Comparison of latent spaces for DML with unit prior (left) and DML with GMM prior containing 4 components (right) for lsdim = 2 on OrganAMNIST dataset. The gaussian components are shown as black with the raidus equal to variance (1). There appears to be no evidence of the distinct gaussian components in the latent space on the right. It does appear that the unit prior may regularize the magnitude of the latent vectors*

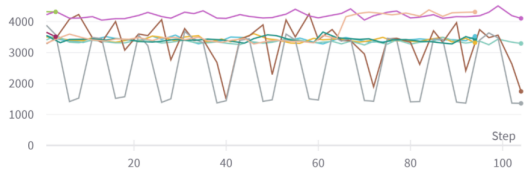


Fig. 6: *Graph of reconstruction loss (componenet of unsupervised loss) of MVAE across epochs. The unsupervised loss does not converge despite being trained on each epoch.*

Architecture Parameters		MVAE Clasif. Accuracy (%)			DML Clasif. Accuracy (%)		
lsdim	gamma	linear	rf	kNN	linear	rf	kNN
2	0	57.8	65.2	64.6	95.0	94.5	95.0
	2	**75.0**	69.8	70.6			
	5	68.8	67.9	66.9			
	10	51.6	**75.4**	**74.2**			
5	0	81.3	83.9	84.5	98.1	97.9	97.9
	2	85.9	88.5	88.4			
	5	82.8	88.0	88.5			
	10	**89.1**	**92.0**	**92.4**			
10	0	85.9	92.1	92.7	98.5	97.7	98.0
	2	87.4	91.8	91.3			
	5	**98.4**	**92.8**	**93.4**			
	10	93.8	92.5	91.9			

Fig. 7: *Table 4: Experiments performed on MVAE architecture across fully labelled MNIST dataset that trains on objective function $L = LU + \gamma * LS$ on fully supervised dataset. The best results for the classification accuracy on the MVAE embeddings in a given latent-dimensionality are bolded.*

[DHS21] Ujjal Kr Dutta, Mehrtash Harandi, and Chellu Chandra Sekhar. Semi-supervised metric learning: A deep resurrection. 2021. doi:10.48550/arXiv.2105.05061.

[EKGB16] Scott Emmons, Stephen Kobourov, Mike Gallant, and Katy Börner. Analysis of network clustering algorithms and cluster quality metrics at scale. *PloS one*, 11(7):e0159161, 2016. doi:10.1371/journal.pone.0159161.

[GTM+21] Antoine Grosnit, Rasul Tutunov, Alexandre Max Maraval, Ryan-Rhys Griffiths, Alexander I Cowen-Rivers, Lin Yang, Lin Zhu, Wenlong Lyu, Zhitang Chen, Jun Wang, et al. High-dimensional bayesian optimisation with variational autoencoders and deep metric learning. *arXiv preprint arXiv:2106.03609*, 2021. doi:10.48550/arXiv.2106.03609.

[KCJ20] Ajinkya Kulkarni, Vincent Colotte, and Denis Jouvet. Deep variational metric learning for transfer of expressivity in multi-speaker text to speech. In *International Conference on Statistical Language and Speech Processing*, pages 157–168. Springer, 2020. doi:10.1007/978-3-030-59430-5_13.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL: http://yann.lecun.com/exdb/mnist/ [cited 2016-01-14 14:24:11].

[LDD+18] Xudong Lin, Yueqi Duan, Qiyuan Dong, Jiwen Lu, and Jie Zhou. Deep variational metric learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 689–704, 2018. doi:10.1007/978-3-030-01267-0_42.

[LYZ+19] Xiaocui Li, Hongzhi Yin, Ke Zhou, Hongxu Chen, Shazia Sadiq, and Xiaofang Zhou. Semi-supervised clustering with deep metric learning. In *International Conference on Database Systems for Advanced Applications*, pages 383–386. Springer, 2019. doi:10.1007/978-3-030-18590-9_50.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RHD+19] Yazhou Ren, Kangrong Hu, Xinyi Dai, Lili Pan, Steven CH Hoi, and Zenglin Xu. Semi-supervised deep embedded clustering. *Neurocomputing*, 325:121–130, 2019. doi:10.1016/j.neucom.2018.10.016.

[SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015. doi:10.1109/cvpr.2015.7298682.

[SLY15] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems*, 28:3483–3491, 2015.

[Soh16] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. In *Advances in neural information processing systems*, pages 1857–1865, 2016.

[WFZ20] Sanyou Wu, Xingdong Feng, and Fan Zhou. Metric learning by similarity network for deep semi-supervised learning. In *Developments of Artificial Intelligence Technologies in Computation and Robotics: Proceedings of the 14th International FLINS Conference (FLINS 2020)*, pages 995–1002. World Scientific, 2020. doi:10.1142/9789811223334_0120.

[WYF13] Qianying Wang, Pong C Yuen, and Guocan Feng. Semi-supervised metric learning via topology preserving multiple semi-supervised assumptions. *Pattern Recognition*, 46(9):2576–2587, 2013. doi:10.1016/j.patcog.2013.02.015.

[YSW+21] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. Medmnist v2: A large-scale lightweight benchmark for 2d and 3d biomedical image classification. *arXiv preprint arXiv:2110.14795*, 2021. doi:10.48550/arXiv.2110.14795.

[ZG21] Zhen Zhu and Yuan Gao. Finding cross-border collaborative centres in biopharma patent networks: A clustering comparison approach based on adjusted mutual information. In *International Conference on Complex Networks and Their Applications*, pages 62–72. Springer, 2021. doi:10.1007/978-3-030-93409-5_6.

[ZRS+20] Meekail Zain, Sonia Rao, Nathan Safir, Quinn Wyner, Isabella Humphrey, Alexa Eldridge, Chenxiao Li, BahaaEddin AlAila, and Shannon P. Quinn. Towards an unsupervised spatiotemporal representation of cilia video using a modular generative pipeline. 2020. doi:10.25080/majora-342d178e-017.

A Python Pipeline for Rapid Application Development (RAD)

Scott D. Christensen‡*, Marvin S. Brown‡, Robert B. Haehnel‡, Joshua Q. Church‡, Amanda Catlett‡, Dallon C. Schofield‡, Quyen T. Brannon‡, Stacy T. Smith‡

Abstract—Rapid Application Development (RAD) is the ability to rapidly prototype an interactive interface through frequent feedback, so that it can be quickly deployed and delivered to stakeholders and customers. RAD is a critical capability needed to meet the ever-evolving demands in scientific research and data science. To further this capability in the Python ecosystem, we have curated and developed a set of open-source tools, including Panel, Bokeh, and Tethys Platform. These tools enable prototyping interfaces in a Jupyter Notebook and facilitate the progression of the interface into a fully-featured, deployable web-application.

Index Terms—web app, Panel, Tethys, Tethys Platform, Bokeh, Jupyter

Introduction

With the tools for data science continually improving and an almost innumerable supply of new data sources, there are seemingly endless opportunities to create new insights and decision support systems. Yet, an investment of resources are needed to extract the value from data using new and improved tools. Well-timed and impactful proposals are necessary to gain the support and resources needed from stakeholders and decision makers to pursue these opportunities. The ability to rapidly prototype capabilities and new ideas provides a powerful visual tool to communicate the impact of a proposal. Interactive applications are even more impactful by engaging the user in the data analysis process.

After a prototype is implemented to communicate ideas and feasibility of a project, additional success is determined by the ability to produce the end product on time and within budget. If the deployable product needs to be completely re-written using different tools, programing languages, and/or frameworks from the prototype, then significantly more time and resources are required. The ability to quickly mature a prototype to production-ready application using the same tool stack can make the difference in the success of a project.

Background

At the US Army Engineer Research and Development Center (ERDC) there are evolving needs to support the missions of the US Army Corps of Engineers and our partners. The scientific Python ecosystem provides a rich set of tools that can be applied to various data sources to provide valuable insights. These insitghts can be integrated into decision support systems that can enhance the information available when making mission critical decisions. Yet, while the opportunities are vast, the ability to get the resources necessary to pursue those opportunities requires effective and timely communication of the value and feasibility of a proposed project.

We have found that rapid prototyping is a very impactful way to concretely show the value that can be obtained from a proposal. Moreover, it also illustrates with clarity that the project is feasible and likely to succeed. Many scientific workflows are developed in Python, and often the prototyping phase is done in a Jupyter Notebook. The Jupyter environment provides an easy way to quickly modify code and visualize output. However, the visualizations are interlaced with the code and thus it does not serve as an ideal way demonstrate the prototype to stakeholders, that may not be familiar with Jupyter Notebooks or code. The Jupyter Dashboard project was addressing this issue before support for it was dropped in 2017. To address this technical gap, we worked with the Holoviz team to develop the Panel library. [Panel] Panel is a high-level Python library for developing apps and dashboards. It enables building layouts with interactive widgets in a Jupyter Notebook environment, but can then easily transition to serving the same code on a standalone secure webserver. This capability enabled us to rapidly prototype workflows and dashboards that could be directly accessed by potential sponsors.

Panel makes prototyping and deploying simple. It can also be iterative. As new features are developed we can continue to work in the Jupyter Notebook environment and then seamlessly transition the new code to a deployed application. Since applications continue to mature they often require production-level features. Panel apps are deployed via Bokeh, and the Bokeh framework lacks some aspects that are needed in some production applications (e.g. a user management system for authentication and permissions, and a database to persist data beyond a session). Bokeh doesn't provide either of these aspects natively.

Tethys Platform is a Django-based web framework that is geared toward making scientific web applications easier to develop by scientists and engineers. [Swain] It provides a Python Software Development Kit (SDK) that enables web apps to be created almost purely in Python, while still leaving the flexibility to add custom HTML, JavaScript, and CSS. Tethys provides user management and role-based permissions control. It also enables database persistence and computational job management

* Corresponding author: Scott.D.Christensen@usace.army.mil
‡ US Army Engineer Research and Development Center

[Christensen], in addition to many visualization tools. Tethys offers the power of a fully-featured web framework without the need to be an expert in full-stack web development. However, Tethys lacks the ease of prototyping in a Jupyter Notebook environment that is provided by Panel.

To support both the rapid prototyping capability provided by Panel and the production-level features of Tethys Platform, we needed a pipeline that could take our Panel-based code and integrate it into the Tethys Platform framework. Through collaborations with the Bokeh development team and developers at Aquaveo, LLC, we were able to create that integration of Panel (Bokeh) and Tethys. This paper demonstrates the seamless pipeline that facilitates Rapid Application Development (RAD). In the next section we describe how the RAD pipeline is used at the ERDC for a particular use case, but first we will provide some background on the use case itself.

Use Case

Helios is a computational fluid dynamics (CFD) code for simulating rotorcraft. It is very computationally demanding and requires High Performance Computing (HPC) resources to execute anything but the most basic of models. At the ERDC we often face a need to run parameter sweeps to determine the affects of varying a particular parameter (or set of parameters). Setting up a Helios model to run on the HPC is a somewhat involved process that requires file management and creating a script to submit the job to the queueing system. When executing a parameter sweep the process becomes even more cumbersome, and is often avoided.

While tedeous to perform manually, the process of modifying input files, transferring to the HPC, and generating and submitting job scripts to the the HPC queueing system can be automated with Python. Furthermore, it can be made much more accessible, even to those without extensive knowledge of how Helios works, through a web-based interface.

Methods

To automate the process of submitting Helios model parameter sweeps to the HPC via a simple interactive web application we developed and used the RAD pipeline. Initially three Helios parameter sweep workflows were identified:

1) Collective Sweep
2) Speed Sweep
3) Ensemble Analysis

The process of submitting each of these workflows to the HPC was similar. They each involved the same basic steps:

1) Authentication to the HPC
2) Connecting to a specific HPC system
3) Specifying the parameter sweep inputs
4) Submtting the job to the queuing system
5) Monitoring the job as it runs
6) Visualizing the results

In fact, these steps are essentially the same for any job being submitted to the HPC. To ensure that we were able to resuse as much code as possible we created PyUIT, a generic, open-source Python library that enables this workflow. The ability to authenticate and connect to the DoD HPC systems is enabled by a service called User Interface Toolkit Plus (UIT+). [PyUIT] UIT+ provides an OAuth2 authentication service and a RESTful

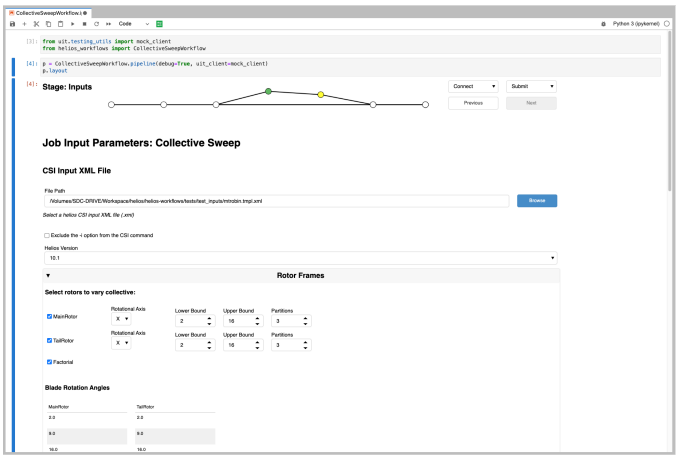


Fig. 1: *Collective Sweep Inputs Stage rendered in a Jupyter Notebook.*

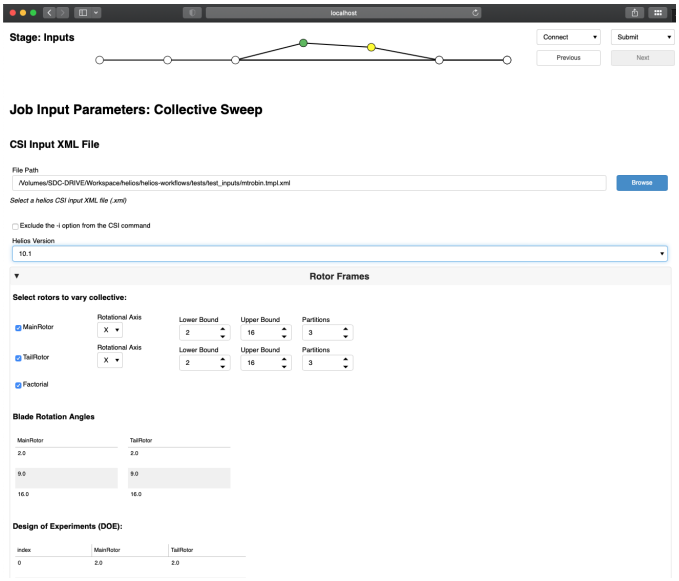


Fig. 2: *Collective Sweep Inputs Stage rendered as a stand-alone Bokeh app.*

API to execute commands on the login nodes of the DoD HPC systems. The PyUIT library provides a Python wrapper for the UIT+ REST API. Additionally, it provides Panel-based interfaces for each of the workflow steps listed above. Panel refers to a workflow comprised of a sequence of steps as a pipeline, and each step in the pipeline is called a stage. Thus, PyUIT provides a template stage class for each step in the basisc HPC workflow.

The PyUIT pipeline stages were customized to create interfaces for each of the three Helios workflows. Other than the inputs stage, the rest of the stages are the same for each of the workflows (See figures 1, 2, and 3). The inputs stage allows the user to select a Helios input file and then provides inputs to allow the user to specify the values for the parameter(s) that will be varied in the sweep. Each of these stages was first created in a Jupyter Notebook. We were then able to deploy each workflow as a standalone Bokeh application. Finally we integrated the Panel-based app into Tethys to leverage the compute job management system and single-sign-on authentication.

As additional features are required, we are able to leverage

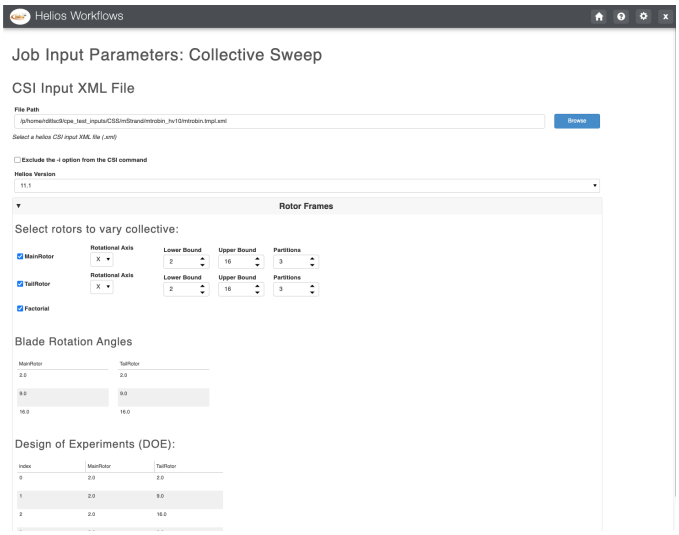


Fig. 3: *Collective Sweep Inputs Stage rendered in the Helios Tethys App.*

the same pipeline: first developing the capability in a Jupyter Notebook, then testing with a Bokeh-served app, and finally, a full integration into Tethys.

Results

By integrating the Panel workflows into the Helios Tethys app we can take advantage of Tethys Platform features, such as the jobs table, which persists metadata about computational jobs in a database.

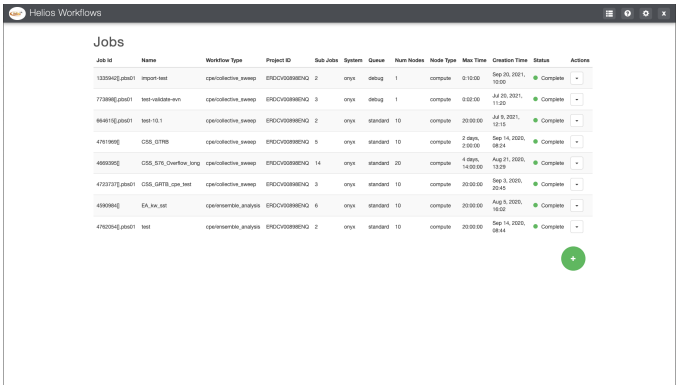


Fig. 4: *Helios Tethys App home page showing a table of previously submitted Helios simulations.*

Each of the three workflows can be launched from the home page of the Helios Tethys app as shown in Figure 5. Although the home page was created in the Tethys framework, once the workflows are launched the same Panel code that was previously developed is called to display the workflow (refer to figures 1, 2, and 3).

From the Tethys Jobs Table different actions are available for each job including viewing results once the job has completed (see 6).

View job results is much more natural in the Tethys app. Helios jobs often take multiple days to complete. By embedding the Helios Panel workflows in Tethys users can leave the web app (ending their session), and then come back later and pull up the

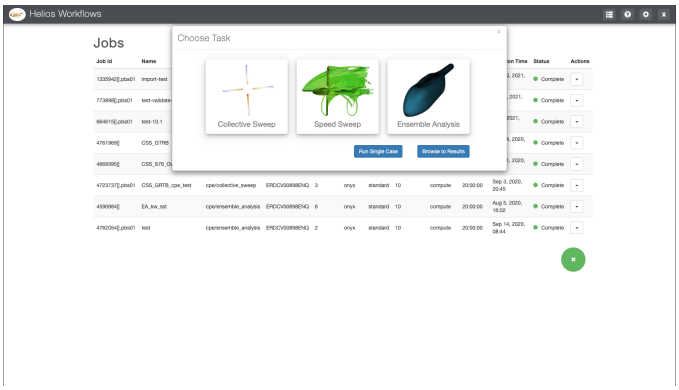


Fig. 5: *The Helios Tethys App is the framework for launching each of the three Panel-based Helios parameter sweep workflows.*

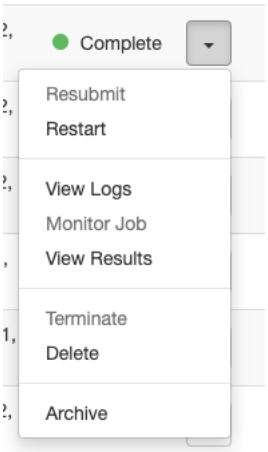


Fig. 6: *Actions associated with a job. The available actions depend on the job's status.*

results to view. The pages that display the results are built with Panel, but Tethys enables them to be populated with information about the job from the database. Figure 7 shows the Tracking Data tab of the results viewer page. The plot is a dynamic Bokeh plot that enables the user to select the data to plot on each axis. This particular plot is showing the variation of the coeffient of drag of the fuselage body over the simulation time.

Figure 8 shows what is called CoViz data, or data that is extracted from the solution as the model is running. This image is showing an isosurface colored by density.

Conclusion

The Helios Tethys App has demonstrated the value of the RAD pipline, which enables both rapid prototyping and rapid progression to production. This enables researchers to quickly communicate and prove ideas and deliver successful products on time. In addition to the Helios Tethys App, RAD has been instrumental for the mission success of various projects at the ERDC.

References

[Christensen] Christensen, S. D., Swain, N. R., Jones, N. L., Nelson, E. J., Snow, A. D., & Dolder, H. G. (2017). A Comprehensive Python Toolkit for Accessing High-Throughput Computing to Support Large Hydrologic Modeling Tasks. JAWRA Journal of the American Water Resources Association, 53(2), 333-343. https://doi.org/10.1111/1752-1688.12455

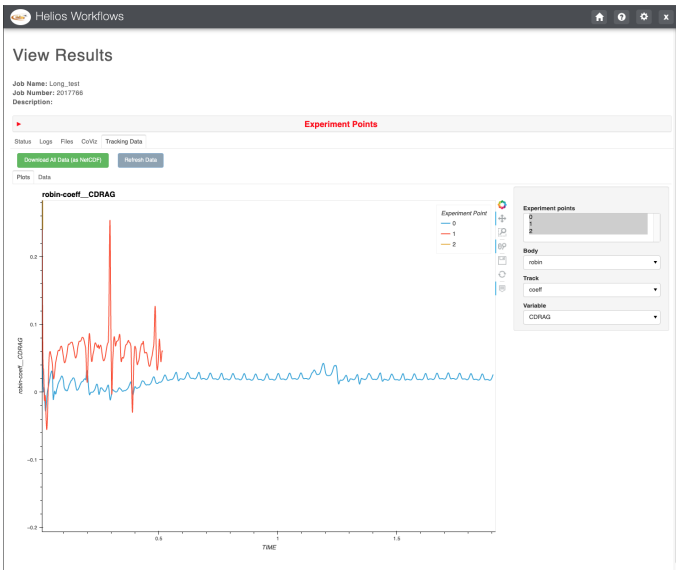


Fig. 7: *Timeseries output associated with a Helios Speed Sweep run.*

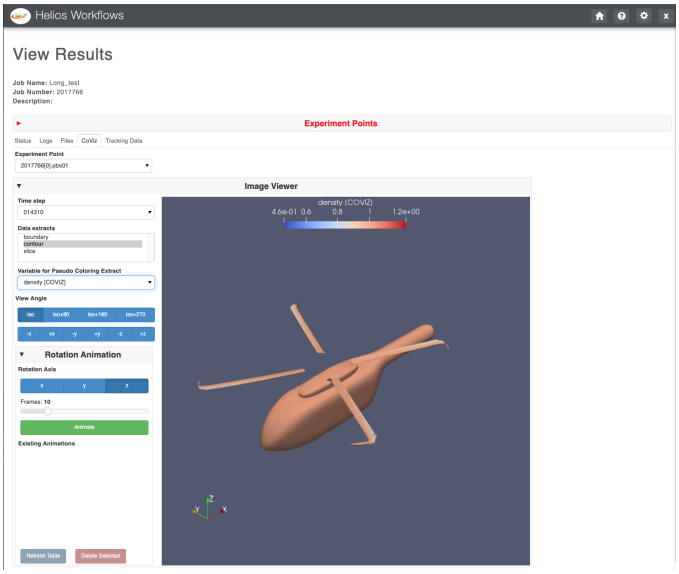


Fig. 8: *Isosurface visualization from a Helios Speed Sweep run.*

[Panel] https://www.panel.org

[PyUIT] https://github.com/erdc/pyuit

[Swain] Swain, N. R., Christensen, S. D., Snow, A. D., Dolder, H., Espinoza-Dávalos, G., Goharian, E., Jones, N. L., Ames, D.P., & Burian, S. J. (2016). A new open source platform for lowering the barrier for environmental web app development. Environmental Modelling & Software, 85, 11-26. https://doi.org/10.1016/j.envsoft.2016.08.003

Monaco: A Monte Carlo Library for Performing Uncertainty and Sensitivity Analyses

W. Scott Shambaugh*

Abstract—This paper introduces *monaco*, a Python library for conducting Monte Carlo simulations of computational models, and performing uncertainty analysis (UA) and sensitivity analysis (SA) on the results. UA and SA are critical to effective and responsible use of models in science, engineering, and public policy, however their use is uncommon. By providing a simple, general, and rigorous-by-default library that wraps around existing models, *monaco* makes UA and SA easy and accessible to practitioners with a basic knowledge of statistics.

Index Terms—Monte Carlo, Modeling, Uncertainty Quantification, Uncertainty Analysis, Sensitivity Analysis, Decision-Making, Ensemble Prediction, VARS, D-VARS

Introduction

Computational models form the backbone of decision-making processes in science, engineering, and public policy. However, our increased reliance on these models stands in contrast to the difficulty in understanding them as we add increasing complexity to try and capture ever more of the fine details of real-world interactions. Practitioners will often take the results of their large, complex model as a point estimate, with no knowledge of how uncertain those results are [FST16]. Multiple-scenario modeling (e.g. looking at a worst-case, most-likely, and best-case scenario) is an improvement, but a complete global exploration of the input space is needed. That gives insight into the overall distribution of results (UA) as well as the relative influence of the different input factors on the ouput variance (SA). This complete understanding is critical for effective and responsible use of models in any decision-making process, and policy papers have identified UA and SA as key modeling practices [ALMR20] [EPA09].

Despite the importance of UA and SA, recent literature reviews show that they are uncommon – in 2014 only 1.3% of all published papers [FST16] using modeling performed any SA. And even when performed, best practices are usually lacking – amongst papers which specifically claimed to perform sensitivity analysis, a 2019 review found only 21% performed global (as opposed to local or zero) UA, and 41% performed global SA [SAB+19].

Typically, UA and SA are done using Monte Carlo simulations, for reasons explored in the following section. There are Monte Carlo frameworks available, however existing options are largely domain-specific, focused on narrow sub-problems (i.e. integration), tailored towards training neural nets, or require a deep statistical background to use. See [OGA+20], [RJS+21], and [DSICJ20] for an overview of the currently available Python tools for performing UA and SA. For the domain expert who wants to perform UA and SA on their existing models, there is not an easy tool to do both in a single shot. *monaco* was written to address this gap.

* *Corresponding author: wsshambaugh@gmail.com*

Fig. 1: *The monaco project logo.*

Motivation for Monte Carlo Approach

Mathematical Grounding

Randomized Monte Carlo sampling offers a cure to the curse of dimensionality: consider an investigation of the output from k input factors $y = f(x_1, x_2, ..., x_k)$ where each factor is uniformly sampled between 0 and 1, $x_i \in U[0,1]$. The input space is then a k-dimensional hypercube with volume 1. If each input is varied one at a time (OAT), then the volume V of the convex hull of the sampled points forms a hyperoctahedron with volume $V = \frac{1}{k!}$ (or optimistically, a hypersphere with $V = \frac{\pi^{k/2}}{2^k \Gamma(k/2+1)}$), both of which decrease super-exponentially as k increases. Unless the model is known to be linear, this leaves the input space wholly unexplored. In contrast, the volume of the convex hull of $n \to \infty$ random samples as is obtained with a Monte Carlo approach will converge to $V = 1$, with much better coverage within that volume as well [DFM92]. See Fig. 2.

Benefits and Drawbacks of Basic Monte Carlo Sampling

monaco focuses on forward uncertainty propagation with basic Monte Carlo sampling. This has several benefits:

- The method is conceptually simple, lowering the barrier of entry and increasing the ease of communicating results to a broader audience.
- The same sample points can be used for UA and SA. Generally, Bayesian methods such as Markov Chain Monte Carlo provide much faster convergence on UA quantities of interest, but their undersampling of regions that do not contribute to the desired quantities is inadequate for SA and complete exploration of the input space. The author's

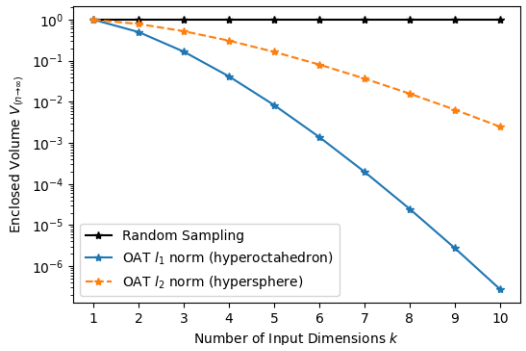


Fig. 2: *Volume fraction V of a k-dimensional hypercube enclosed by the convex hull of $n \to \infty$ random samples versus OAT samples along the principle axes of the input space.*

experience aligns with [SAB+19] in that there is great practical benefit in broad sampling without pigeonholing one's purview to particular posteriors, through uncovering bugs and edge cases in regions of input space that were not being previously considered.

- It can be applied to domains that are not data-rich. See for example NASA's use of Monte Carlo simulations during rocket design prior to collecting test flight data [HB10].

However, basic Monte Carlo sampling is subject to the classical drawbacks of the method such as poor sampling of rare events and the slow $\sigma/\sqrt{n}$ convergence on quantities of interest. If the outputs and regions of interest are firmly known at the outset, then other sampling methods will be more efficient [KTB13].

Additionally, given that any conclusions are conditional on the correctness of the underlying model and input parameters, the task of validation is critical to confidence in the UA and SA results. However, this is currently out of scope for the library and must be performed with other tools. In a data-poor domain, hypothesis testing or probabilistic prediction measures like loss scores can be used to anchor the outputs against a small number of real-life test data. More generally, the "inverse problem" of model and parameter validation is a deep field unto itself and [C+12] and [SLKW08] are recommended as overviews of some methods. If *monaco*'s scope is too limited for the reader's needs, the author recommends *UQpy* [OGA+20] for UA and SA, and *PyMC* [SWF16] or *Stan* [CGH+17] as good general-purpose probabilistic programming Python libraries.

Workflow

UA and SA of any model follows a common workflow. Probability distributions for the model inputs are defined, and randomly sampled values for a large number of cases are fed to the model. The outputs from each case are collected and the full set of inputs and outputs can be analyzed. Typically, UA is performed by generating histograms, scatter plots, and summary statistics for the output variables, and SA is performed by looking at the effect of input on output variables through scatter plots, performing regressions, and calculating sensitivity indices. These results can then be compared to real-world test data to validate the model or inform revisions to the model and input variables. See Fig. 3.

Note that with model and input parameter validation currently outside *monaco*'s scope, closing that part of the workflow loop is left up to the user.

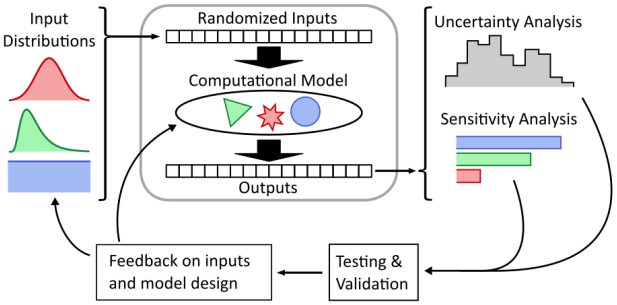


Fig. 3: *Monte Carlo workflow for understanding the full behavior of a computational model, inspired by [SAB+19].*

monaco Structure

Overall Structure

Broadly, each input factor and model output is a *variable* that can be thought of as lists (rows) containing the full range of randomized *values*. *Cases* are slices (columns) that take the *i*'th input and output value for each variable, and represent a single run of the model. Each case is run on its own, and the output values are collected into output variables. Fig. 4 shows a visual representation of this.

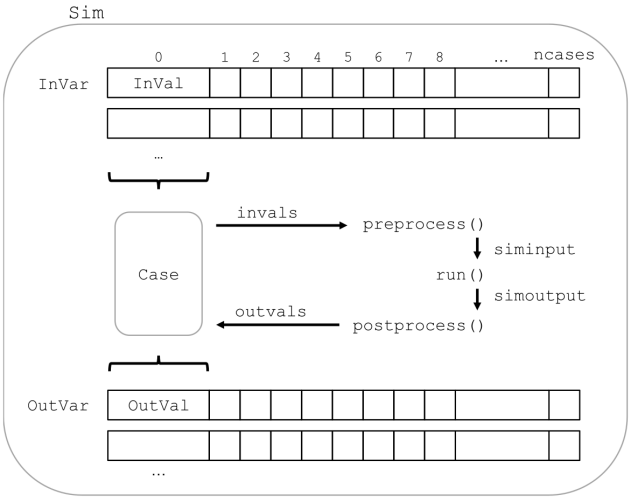


Fig. 4: *Structure of a monaco simulation, showing the relationship between the major objects and functions. This maps onto the central block in Fig. 3.*

Simulation Setup

The base of a *monaco* simulation is the *Sim* object. This object is formed by passing it a name, the number of random cases *ncases*, and a dict *fcns* of the handles for three user-defined functions detailed in the next section. A random seed that then seeds the entire simulation can also be passed in here, and is highly recommended for repeatability of results.

Input variables then need to be defined. *monaco* takes in the handle to any of *scipy.stat*'s continuous or discrete probability distributions, as well as the required arguments for that probability distribution [VGO+20]. If nonnumeric inputs are desired, the method can also take in a *nummap* dictionary which maps the randomly drawn integers to values of other types.

At this point the sim can be run. The randomized drawing of input values, creation of cases, running of those cases, and extraction of output values are automatically executed.

User-Defined Functions

The user needs to define three functions to wrap *monaco*'s Monte Carlo structure around their existing computational model. First is a *run* function which either calls or directly implements their model. Second is a *preprocess* function which takes in a *Case* object, extracts the randomized inputs, and structures them with any other invariant data to pass to the *run* function. Third is a *postprocess* function which takes in a *Case* object as well as the results from the model, and extracts the desired output values. The Python call chain is as:

```
postprocess(case, *run(*preprocess(case)))
```

Or equivalently to expand the Python star notation into pseudocode:

```
siminput = (siminput1, siminput2, ...)
         = preprocess(case)
simoutput = (simoutput1, simoutput2, ...)
          = run(*siminput)
          = run(siminput1, siminput2, ...)
_ = postprocess(case, *simoutput)
  = postprocess(case, simoutput1, simoutput2, ...)
```

These three functions must be passed to the simulation in a dict with keys *'run'*, *'preprocess'*, and *'postprocess'*. See the example code at the end of the paper for a simple worked example.

Examining Results

After running, users should generally do all of the following UA and SA tasks to get a full picture of the behavior of their computational model.

- Plot the results (UA & SA).
- Calculate statistics for input or output variables (UA).
- Calculate sensitivity indices to rank importance of the input variables on variance of the output variables (SA).
- Investigate specific cases with outlier or puzzling results.
- Save the results to file or pass them to other programs.

Data Flow

A summary of the process and data flow:

1) Instantiate a *Sim* object.
2) Add input variables to the sim with specified probability distributions.
3) Run the simulation. This executes the following:
    a) Random percentiles $p_i \in U[0,1]$ are drawn *ndraws* times for each of the input variables.
    b) These percentiles are transformed into random values via the inverse cumulative density function of the target probability distribution $x_i = F^{-1}(p_i)$.
    c) If nonnumeric inputs are desired, the numbers are converted to objects via a *nummap* dict.
    d) *Case* objects are created and populated with the input values for each case.
    e) Each case is run by structuring the inputs values with the *preprocess* function, passing them to the *run* function, and collecting the output values with the *postprocess* function.
    f) The output values are collected into output variables and saved back to the sim. If the values are nonnumeric, a *valmap* dict assigning numbers to each unique value is automatically generated.
4) Calculate statistics & sensitivities for input & output variables.
5) Plot variables, their statistics, and sensitivities.

Incorporating into Existing Workflows

If the user wants to use existing workflows for generating, running, post-processing, or examining results, any combination of *monaco*'s major steps can be replaced with external tooling by saving and loading input and output variables to file. For example, *monaco* can be used only for its parallel processing backend by importing existing randomly drawn input variables, running the simulation, and exporting the output variables for outside analysis. Or, it can be used only for its plotting and analysis capabilities by feeding it inputs and outputs generated elsewhere.

Resource Usage

Note that *monaco*'s computational and storage overhead in creating easily-interrogatable objects for each variable, value, and case makes it an inefficient choice for computationally simple applications with high n, such as Monte Carlo integration. If the preprocessed sim input and raw output for each case (which for some models may dominate storage) is not retained, then the storage bottleneck will be the creation of a *Val* object for each case's input and output values with minimum size 0.5 kB. The maximum n will be driven by the size of the RAM on the host machine being capable of holding at least $0.5 * n(k_{in} + k_{out})$ kB. On the computational bottleneck side, *monaco* is best suited for models where the model runtime dominates the random variate generation and the few hundred microseconds of *dask.delayed* task switching time.

Technical Features

Sampling Methods

Random sampling of the percentiles for each variable can be done using scipy's pseudo-random number generator (PRNG), or with any of the low-discrepancy methods from the *scip.stats.qmc* quasi-Monte Carlo (QMC) module. QMC in general provides faster $O(\log(n)^k n^{-1})$ convergence compared to the $O(n^{-1/2})$ convergence of random sampling [Caf98]. Available low-discrepancy options are regular or scrambled Sobol sequences, regular or scrambled Halton sequences, or Latin Hypercube Sampling. In general, the *'sobol_random'* method that generates scrambled Sobol sequences [Sob67] [Owe20] is recommended in nearly all cases as the sequence with the fastest QMC convergence [CKK18], balanced integration properties as long as the number of cases is a power of 2, and a fairly flat frequency spectrum (though sampling spectra are rarely a concern) [PCX+18]. See Fig. 5 for a visual comparison of some of the options.

Order Statistics, or, How Many Cases to Run?

How many Monte Carlo cases should one run? One answer would be to choose $n \geq 2^k$ with a sampling method that implements a (t,m,s) digital net (such as a Sobol or Halton sequence), which guarantees that there will be at least one sample point in every hyperoctant of the input space [JK08]. This should be considered a lower bound for SA, with the number of cases run being some integer multiple of 2^k.

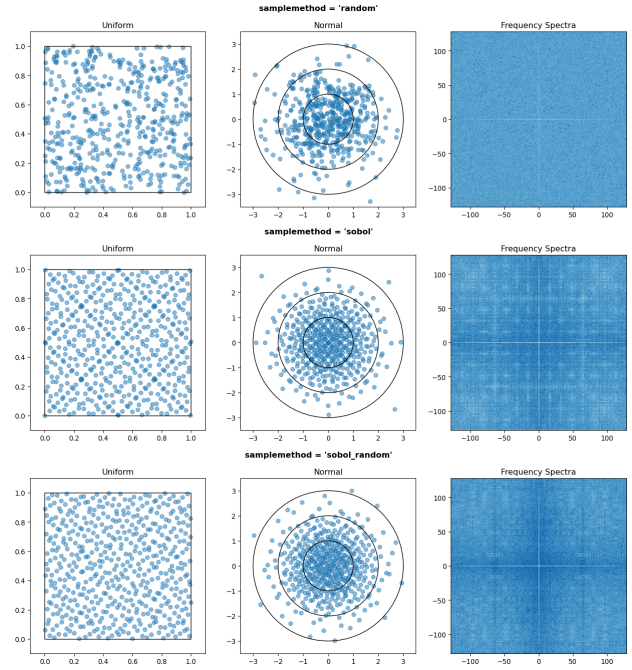


Fig. 5: *256 uniform and normal samples along with the 2D frequency spectra for PRNG random sampling (top), Sobol sampling (middle), and scrambled Sobol sampling (bottom, default).*

Along a similar vein, [DFM92] suggests that with random sampling $n \geq 2.136^k$ is sufficient to ensure that the volume fraction V approaches 1. The author hypothesizes that for a digital net, the $n \geq \lambda^k$ condition will be satisfied with some $\lambda \leq 2$, and so $n \geq 2^k$ will suffice for this condition to hold. However, these methods of choosing the number of cases may undersample for low k and be infeasible for high k.

A rigorous way of choosing the number of cases is to first choose a statistical interval (e.g. a confidence interval for a percentile, or a tolerance interval to contain a percent of the population), and then use order statistics to calculate the minimum n required to obtain that result at a desired confidence level. This approach is independent of k, making UA of high-dimensional models tractable. *monaco* implements order statistics routines for calculating these statistical intervals with a distribution-free approach that makes no assumptions about the normality or other shape characteristics of the output distribution. See Chapter 5 of [HM91] for background.

A more qualitative UA method would simply be to choose a reasonably high n (say, $n = 2^{10}$), manually examine the results to ensure high-interest areas are not being undersampled, and rely on bootstrapping of the desired variable statistics to obtain the required confidence levels.

Variable Statistics

For any input or output variable, a statistic can be calculated for the ensemble of values. *monaco* builds in some common statistics (mean, percentile, etc), or alternatively the user can pass in a custom one. To obtain a confidence interval for this statistic, the results are resampled with replacement using the *scipy.stats.bootstrap* module. The number of bootstrap samples is determined using an order statistic approach as outlined in the previous section, and multiplying that number by a scaling factor (default 10x) for smoothness of results.

Sensitivity Indices

Sensitivity indices give a measure of the relationship between the variance of a scalar output variable to the variance of each of the input variables. In other words, they measure which of the input ranges have the largest effect on an output range. It is crucial that sensitivity indices are global rather than local measures – global sensitivity has the stronger theoretical grounding and there is no reason to rely on local measures in scenarios such as automated computer experiments where data can be easily and arbitrarily sampled [SRA+08] [PBPS22].

With computer-designed experiments, it is possible to construct a specially constructed sample set to directly calculate global sensitivity indices such as the Total-Order Sobol index [Sob01], or the IVARS100 index [RG16]. However, this special construction requires either sacrificing the desirable UA properties of low-discrepancy sampling, or conducting an additional Monte Carlo analysis of the model with a different sample set. For this reason, *monaco* uses the D-VARS approach to calculating global sensitivity indices, which allows for using a set of given data [SR20]. This is the first publically available implementation of the D-VARS algorithm.

Plotting

monaco includes a plotting module that takes in input and output variables and quickly creates histograms, empirical CDFs, scatter plots, or 2D or 3D "spaghetti plots" depending on what is most appropriate for each variable. Variable statistics and their confidence intervals are automatically shown on plots when applicable.

Vector Data

If the values for an output variable are length s lists, NumPy arrays, or Pandas dataframes, they are treated as timeseries with s steps. Variable statistics for these variables are calculated on the ensemble of values at each step, giving time-varying statistics.

The plotting module will automatically plot size $(1,s)$ arrays against the step number as 2-D lines, size $(2,s)$ arrays as 2-D parametric lines, and size $(3,s)$ arrays as 3-D parametric lines.

Parallel Processing

monaco uses *dask.distributed* [Roc15] as a parallel processing backend, and supports preprocessing, running, and postprocessing cases in a parallel arrangement. Users familiar with *dask* can extend the parallelization of their simulation from their single machine to a distributed cluster.

For simple simulations such as the example code at the end of the paper, the overhead of setting up a *dask* server may outweigh the speedup from parallel computation, and in those cases *monaco* also supports running single-threaded in a single for-loop.

The Median Case

A "nominal" run is often useful as a baseline to compare other cases against. If desired, the user can set a flag to force the first case to be the median 50th percentile draw of all the input variables prior to random sampling.

Debugging Cases

By default, all the raw results from each case's simulation run prior to postprocessing are saved to the corresponding *Case* object. Individual cases can be interrogated by looking at these raw results, or by indicating that their results should be highlighted

in plots. If some cases fail to run, *monaco* will mark them as incomplete and those specific cases can be rerun without requiring the full set of cases to be recomputed. A *debug* flag can be set to not skip over failed cases and instead stop at a breakpoint or dump the stack trace on encountering an exception.

Saving and Loading to File

The base *Sim* object and the *Case* objects can be serialized and saved to or loaded from *.mcsim* and *.mccase* files respectively, which are stored in a results directory. The Case objects are saved separately since the raw results from a run of the simulation may be arbitrarily large, and the Sim object can be comparatively lightweight. Loading the Sim object from file will automatically attempt to load the cases in the same directory, but can also stand alone if the raw results are not needed.

Alternatively, the numerical representations for input and output variables can be saved to and loaded from *.json* or *.csv* files. This is useful for interfacing with external tooling, but discards the metadata that would be present by saving to *monaco*'s native objects.

Example

Presented here is a simple example showing a Monte Carlo simulation of rolling two 6-sided dice and looking at their sum.

The user starts with their *run* function which here directly implements their computational model. They must then create *preprocess* and *postprocess* functions to feed in the randomized input values and collect the outputs from that model.

```
# The 'run' function, which implements the
# existing computational model (or wraps it)
def example_run(die1, die2):
    dicesum = die1 + die2
    return (dicesum, )

# The 'preprocess' function grabs the random
# input values for each case and structures it
# with any other data in the format the 'run'
# function expects
def example_preprocess(case):
    die1 = case.invals['die1'].val
    die2 = case.invals['die2'].val
    return (die1, die2)

# The 'postprocess' function takes the output
# from the 'run' function and saves off the
# outputs for each case
def example_postprocess(case, dicesum):
    case.addOutVal(name='Sum', val=dicesum)
    case.addOutVal(name='Roll Number',
                   val=case.ncase)
    return None
```

The *monaco* simulation is initialized, given input variables with specified probability distributions (here a random integer between 1 and 6), and run.

```
import monaco as mc
from scipy.stats import randint

# dict structure for the three input functions
fcns = {'run'        : example_run,
        'preprocess' : example_preprocess,
        'postprocess': example_postprocess}

# Initialize the simulation
ndraws = 1024  # Arbitrary for this example
seed = 123456  # Recommended for repeatability

sim = mc.Sim(name='Dice Roll', ndraws=ndraws,
             fcns=fcns, seed=seed)

# Generate the input variables
sim.addInVar(name='die1', dist=randint,
             distkwargs={'low': 1, 'high': 6+1})
sim.addInVar(name='die2', dist=randint,
             distkwargs={'low': 1, 'high': 6+1})

# Run the Simulation
sim.runSim()
```

The results of the simulation can then be analyzed and examined. Fig. 6 shows the plots this code generates.

```
# Calculate the mean and 5-95th percentile
# statistics for the dice sum
sim.outvars['Sum'].addVarStat('mean')
sim.outvars['Sum'].addVarStat('percentile',
                              {'p':[0.05, 0.95]})

# Plots a histogram of the dice sum
mc.plot(sim.outvars['Sum'])

# Creates a scatter plot of the sum vs the roll
# number, showing randomness
mc.plot(sim.outvars['Sum'],
        sim.outvars['Roll Number'])

# Calculate the sensitivity of the dice sum to
# each of the input variables
sim.calcSensitivities('Sum')
sim.outvars['Sum'].plotSensitivities()
```

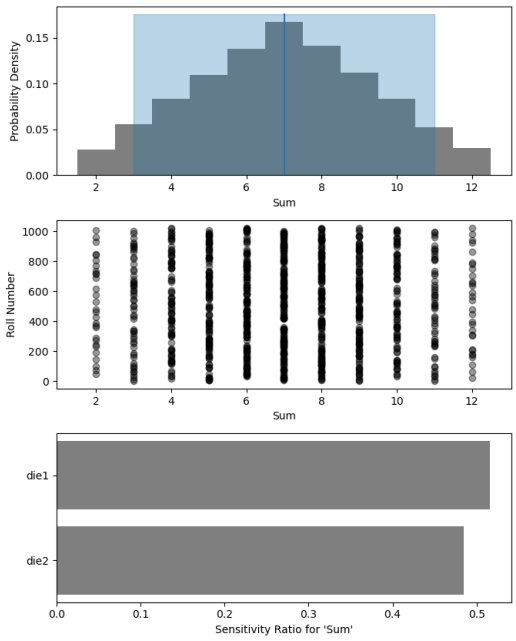


Fig. 6: *Output from the example code which calculates the sum of two random dice rolls. The top plot shows a histogram of the 2-dice sum with the mean and 5–95th percentiles marked, the middle plot shows the randomness over the set of rolls, and the bottom plot shows that each of the dice contributes 50% to the variance of the sum.*

Case Studies

These two case studies are toy models meant as illustrative of potential uses, and not of expertise or rigor in their respective domains. Please see https://github.com/scottshambaugh/monaco/tree/main/examples for their source code as well as several more Monte Carlo implementation examples across a range of domains including financial modeling, pandemic spread, and integration.

Baseball

This case study models the trajectory of a baseball in flight after being hit for varying speeds, angles, topspins, aerodynamic conditions, and mass properties. From assumed initial conditions immediately after being hit, the physics of the ball's ballistic flight are calculated over time until it hits the ground.

Fig. 7 shows some plots of the results. A baseball team might use analyses like this to determine where outfielders should be placed to catch a ball for a hitter with known characteristics, or determine what aspect of a hit a batter should focus on to improve their home run potential.

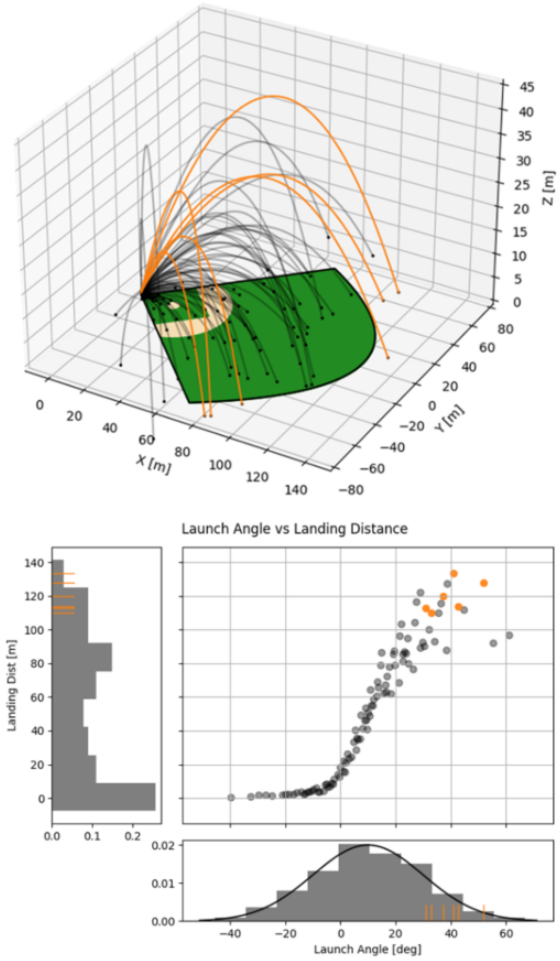


Fig. 7: *100 simulated baseball trajectories (top), and the relationship between launch angle and landing distance (bottom). Home runs are highlighted in orange.*

Election

This case study attempts to predict the result of the 2020 US presidential election, based on polling data from FiveThirtyEight 3 weeks prior to the election [Fiv20].

Each state independently casts a normally distributed percentage of votes for the Democratic, Republican, and Other candidates, based on polling. Also assumed is a uniform ±3% national swing due to polling error which is applied to all states equally. That summed percentage is then normalized so the total for all candidates is 100%. The winner of each state's election assigns their electoral votes to that candidate, and the candidate that wins at least 270 of the 538 electoral votes is the winner.

The calculated win probabilities from this simulation are 93.4% Democratic, 6.2% Republican, and 0.4% Tie. The 25–75th percentile range for the number of electoral votes for the Democratic candidate is 281–412, and the actual election result was 306 electoral votes. See Fig. 8.

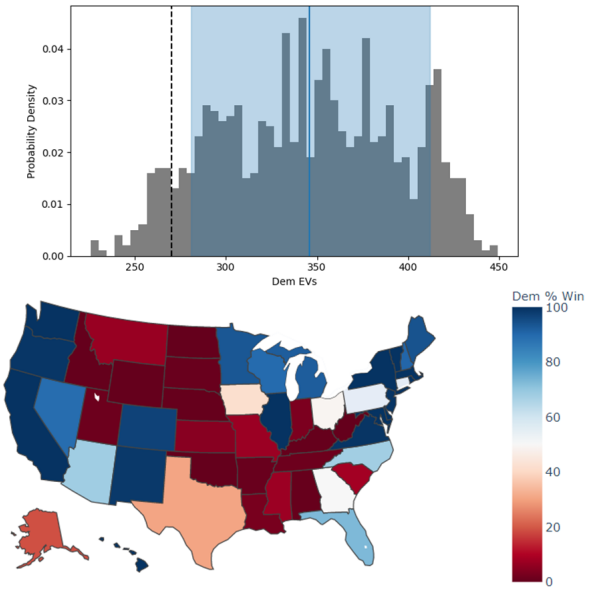


Fig. 8: *Predicted electoral votes for the Democratic 2020 US Presidential candidate with the median and 25-75th percentile interval marked (top), and a map of the predicted Democratic win probability per state (bottom).*

Conclusion

This paper has introduced the ideas underlying Monte Carlo analysis and discussed when it is appropriate to use for conducting UA and SA. It has shown how *monaco* implements a rigorous, parallel Monte Carlo process, and how to use it through a simple example and two case studies. This library is geared towards scientists, engineers, and policy analysts that have a computational model in their domain of expertise, enough statistical knowledge to define a probability distribution, and a desire to ensure their model will make accurate predictions of reality. The author hopes this tool will help contribute to easier and more widespread use of UA and SA in improved decision-making.

Further Information

monaco is available on PyPI as the package `monaco`, has API documentation at https://monaco.rtfd.io/, and is hosted on github at https://github.com/scottshambaugh/monaco/.

References

[ALMR20] I Azzini, G Listorti, TA Mara, and R Rosati. Uncertainty and sensitivity analysis for policy decision making. *An Introductory Guide. Joint Research Centre, European Commission, Luxembourg*, 2020. doi:10.2760/922129.

[C+12] National Research Council et al. *Assessing the reliability of complex models: mathematical and statistical foundations of verification, validation, and uncertainty quantification*. National Academies Press, 2012. doi:10.17226/13395.

[Caf98] Russel E Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 7:1–49, 1998. doi:10.1017/S0962492900002804.

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017. doi:10.18637/jss.v076.i01.

[CKK18] Per Christensen, Andrew Kensler, and Charlie Kilpatrick. Progressive multi-jittered sample sequences. In *Computer Graphics Forum*, volume 37, pages 21–33. Wiley Online Library, 2018. doi:10.1111/cgf.13472.

[DFM92] Martin E. Dyer, Zoltan Füredi, and Colin McDiarmid. Volumes spanned by random points in the hypercube. *Random Structures & Algorithms*, 3(1):91–106, 1992. doi:10.1002/rsa.3240030107.

[DSICJ20] Dominique Douglas-Smith, Takuya Iwanaga, Barry F.W. Croke, and Anthony J. Jakeman. Certain trends in uncertainty and sensitivity analysis: An overview of software tools and techniques. *Environmental Modelling & Software*, 124, 2020. doi:10.1016/j.envsoft.2019.104588.

[EPA09] US EPA. Guidance on the development, evaluation, and application of environmental models (epa/100/k-09/003), 2009. URL: https://nepis.epa.gov/Exe/ZyPDF.cgi?Dockey=P1003E4R.PDF.

[Fiv20] FiveThirtyEight. 2020 general election forecast - state topline polls-plus data, October 2020. URL: https://github.com/fivethirtyeight/data/tree/master/election-forecasts-2020.

[FST16] Federico Ferretti, Andrea Saltelli, and Stefano Tarantola. Trends in sensitivity analysis practice in the last decade. *Science of the total environment*, 568:666–670, 2016. doi:10.1016/j.scitotenv.2016.02.133.

[HB10] John Hanson and Bernard Beard. Applying monte carlo simulation to launch vehicle design and requirements verification. In *AIAA Guidance, Navigation, and Control Conference*. American Institute of Aeronautics and Astronautics, 2010. doi:10.2514/6.2010-8433.

[HM91] Gerald J Hahn and William Q Meeker. *Statistical intervals: a guide for practitioners*. John Wiley & Sons, 1991. doi:10.1002/9780470316771.ch5.

[JK08] Stephen Joe and Frances Y Kuo. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing*, 30(5):2635–2654, 2008. doi:10.1137/070709359.

[KTB13] Dirk P Kroese, Thomas Taimre, and Zdravko I Botev. *Handbook of monte carlo methods*. John Wiley & Sons, 2013. doi:10.1002/9781118014967.

[OGA+20] Audrey Olivier, Dimitris G. Giovanis, B.S. Aakash, Mohit Chauhan, Lohit Vandanapu, and Michael D. Shields. Uqpy: A general purpose python package and development environment for uncertainty quantification. *Journal of Computational Science*, 47:101204, 2020. doi:10.1016/j.jocs.2020.101204.

[Owe20] Art B Owen. On dropping the first sobol'point. *arXiv preprint arXiv:2008.08051*, 2020. doi:10.48550/arXiv.2008.08051.

[PBPS22] Arnald Puy, William Becker, Samuele Lo Piano, and Andrea Saltelli. A comprehensive comparison of total-order estimators for global sensitivity analysis. *International Journal for Uncertainty Quantification*, 12(2), 2022. doi:int.j.uncertaintyquantification.2021038133.

[PCX+18] Hélène Perrier, David Coeurjolly, Feng Xie, Matt Pharr, Pat Hanrahan, and Victor Ostromoukhov. Sequences with low-discrepancy blue-noise 2-d projections. In *Computer Graphics Forum*, volume 37, pages 339–353. Wiley Online Library, 2018. doi:10.1111/cgf.13366.

[RG16] Saman Razavi and Hoshin V Gupta. A new framework for comprehensive, robust, and efficient global sensitivity analysis: 1. theory. *Water Resources Research*, 52(1):423–439, 2016. doi:10.1002/2015wr017558.

[RJS+21] Saman Razavi, Anthony Jakeman, Andrea Saltelli, Clémentine Prieur, Bertrand Iooss, Emanuele Borgonovo, Elmar Plischke, Samuele Lo Piano, Takuya Iwanaga, William Becker, et al. The future of sensitivity analysis: An essential discipline for systems modeling and policy support. *Environmental Modelling & Software*, 137:104954, 2021. doi:10.1016/j.envsoft.2020.104954.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015. doi:10.25080/majora-7b98e3ed-013.

[SAB+19] Andrea Saltelli, Ksenia Aleksankina, William Becker, Pamela Fennell, Federico Ferretti, Niels Holst, Sushan Li, and Qiongli Wu. Why so many published sensitivity analyses are false: A systematic review of sensitivity analysis practices. *Environmental modelling & software*, 114:29–39, 2019. doi:10.1016/j.envsoft.2019.01.012.

[SLKW08] Richard M Shiffrin, Michael D Lee, Woojae Kim, and Eric-Jan Wagenmakers. A survey of model evaluation approaches with a tutorial on hierarchical bayesian methods. *Cognitive Science*, 32(8):1248–1284, 2008. doi:10.1080/03640210802414826.

[Sob67] Ilya M Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967. doi:10.1016/0041-5553(67)90144-9.

[Sob01] Ilya M Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and computers in simulation*, 55(1-3):271–280, 2001. doi:10.1016/s0378-4754(00)00270-6.

[SR20] Razi Sheikholeslami and Saman Razavi. A fresh look at variography: measuring dependence and possible sensitivities across geophysical systems from any given data. *Geophysical Research Letters*, 47(20):e2020GL089829, 2020. doi:10.1029/2020gl089829.

[SRA+08] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008. doi:10.1002/9780470725184.

[SWF16] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016. doi:10.7717/peerj-cs.55.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020. doi:10.14293/s2199-1006.1.sor-life.a7056644.v1.rysreg.

Enabling Active Learning Pedagogy and Insight Mining with a Grammar of Model Analysis

Zachary del Rosario[‡*]

Abstract—Modern engineering models are complex, with dozens of inputs, uncertainties arising from simplifying assumptions, and dense output data. While major strides have been made in the computational scalability of complex models, relatively less attention has been paid to user-friendly, reusable tools to explore and make sense of these models. Grama is a python package aimed at supporting these activities. Grama is a grammar of model analysis: an ontology that specifies data (in tidy form), models (with quantified uncertainties), and the verbs that connect these objects. This definition enables a reusable set of evaluation "verbs" that provide a consistent analysis toolkit across different grama models. This paper presents three case studies that illustrate pedagogy and engineering work with grama: 1. Providing teachable moments through errors for learners, 2. Providing reusable tools to help users self-initiate productive modeling behaviors, and 3. Enabling *exploratory model analysis* (EMA) – exploratory data analysis augmented with data generation.

Index Terms—engineering, engineering education, exploratory model analysis, software design, uncertainty quantification

Introduction

Modern engineering relies on scientific computing. Computational advances enable faster analysis and design cycles by reducing the need for physical experiments. For instance, finite-element analysis enables computational study of aerodynamic flutter, and Reynolds-averaged Navier-Stokes simulation supports the simulation of jet engines. Both of these are enabling technologies that support the design of modern aircraft [KN05]. Modern areas of computational research include heterogeneous computing environments [MV15], task-based parallelism [BTSA12], and big data [SS13]. Another line of work considers the development of *integrated tools* to unite diverse disciplinary perspectives in a single, unified environment (e.g., the integration of multiple physical phenomena in a single code [EVB+20] or the integration of a computational solver and data analysis tools [MTW+22]). Such integrated computational frameworks are highlighted as *essential* for applications such as computational analysis and design of aircraft [SKA+14]. While engineering computation has advanced along the aforementioned axes, the conceptual understanding of practicing engineers has lagged in key areas.

Every aircraft you have ever flown on has been designed using probabilistically-flawed, potentially dangerous criteria [dRFI21]. The fundamental issue underlying these criteria is a flawed heuristic for uncertainty propagation; initial human subjects work suggests that engineers' tendency to misdiagnose sources of variability as inconsequential noise may contribute to the persistent application of flawed design criteria [AFD+21]. These flawed treatments of uncertainty are not limited to engineering design; recent work by Kahneman et al. [KSS21] highlights widespread failures to recognize or address variability in human judgment, leading to bias in hiring, economic loss, and an unacceptably capricious application of justice.

Grama was originally developed to support model analysis under uncertainty; in particular, to enable *active learning* [FEM+14] – a form of teaching characterized by active student engagement shown to be superior to lecture alone. This toolkit aims to *integrate* the disciplinary perspectives of computational engineering and statistical analysis within a unified environment to support a *coding to learn* pedagogy [Bar16] – a teaching philosophy that uses code to teach a discipline, rather than as a means to teach computer science or coding itself. The design of grama is heavily inspired by the Tidyverse [WAB+19], an integrated set of R packages organized around the 'tidy data' concept [Wic14]. Grama uses the tidy data concept and introduces an analogous concepts for *models*.

Grama: A Grammar of Model Analysis

Grama [dR20] is an integrated set of tools for working with *data* and *models*. Pandas [pdt20], [WM10] is used as the underlying data class, while grama implements a `Model` class. A grama model includes a number of functions – mathematical expressions or simulations – and domain/distribution information for the deterministic/random inputs. The following code illustrates a simple grama model with both deterministic and random inputs[1].

```
# Each cp_* function adds information to the model
md_example = (
    gr.Model("An example model")
    # Overloaded `>>` provides pipe syntax
    >> gr.cp_vec_function(
        fun=lambda df: gr.df_make(f=df.x+df.y+df.z),
        var=["x", "y", "z"],
        out=["f"],
    )
    >> gr.cp_bounds(x=(-1, +1))
    >> gr.cp_marginals(
        y=gr.marg_mom("norm", mean=0, sd=1),
        z=gr.marg_mom("uniform", mean=0, sd=1),
    )
```

1. Throughout, `import grama as gr` is assumed.

* Corresponding author: zdelrosario@olin.edu
‡ Assistant Professor of Engineering and Applied Statistics, Olin College of Engineering

```
    >> gr.cp_copula_gaussian(
        df_corr=gr.df_make(
            var1="y",
            var2="z",
            corr=0.5,
        )
    )
)
```

While an engineer's interpretation of the term "model" focuses on the input-to-output mapping (the simulation), and a statistician's interpretation of the term "model" focuses on a distribution, the grama model integrates both perspectives in a single model.

Grama models are intended to be *evaluated* to generate data. The data can then be analyzed using visual and statistical means. Models can be *composed* to add more information, or *fit* to a dataset. Figure 1 illustrates this interplay between data and models in terms of the four categories of function "verbs" provided in grama.

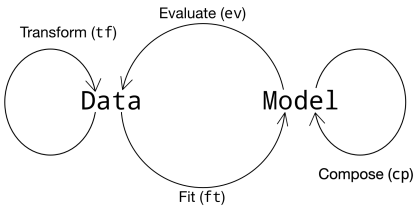


Fig. 1: *Verb categories in grama. These grama functions start with an identifying prefix, e.g. `ev_*` for evaluation verbs.*

Defaults for Concise Code

Grama verbs are designed with sensible default arguments to enable concise code. For instance, the following code visualizes input sweeps across its three inputs, similar to a *ceteris paribus* profile [KBB19], [Bie20].

```
(
    ## Concise default analysis
    md_example
    >> gr.ev_sinews(df_det="swp")
    >> gr.pt_auto()
)
```

This code uses the default number of sweeps and sweep density, and constructs a visualization of the results. The resulting plot is shown in Figure 2.

Grama imports the plotnine package for data visualization [HK21], both to provide an expressive grammar of graphics, but also to implement a variety of "autoplot" routines. These are called via a dispatcher `gr.pt_auto()` which uses metadata from evaluation verbs to construct a default visual. Combined with sensible defaults for keyword arguments, these tools provide a concise syntax even for sophisticated analyses. The same code can be slightly modified to change a default argument value, or to use plotnine to create a more tailored visual.

```
(
    md_example
    ## Override default parameters
    >> gr.ev_sinews(df_det="swp", n_sweeps=10)
    >> gr.pt_auto()
)

(
    md_example
    >> gr.ev_sinews(df_det="swp")
    ## Construct a targeted plot
```

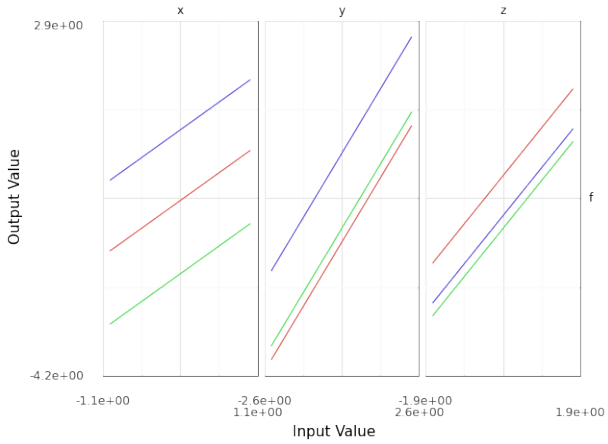


Fig. 2: *Input sweep generated from the code above. Each panel visualizes the effect of changing a single input, with all other inputs held constant.*

```
    >> gr.tf_filter(DF.sweep_var == "x")
    >> gr.ggplot(gr.aes("x", "f", group="sweep_ind"))
    + gr.geom_line()
)
```

This system of defaults is important for pedagogical design: Introductory grama code can be made extremely simple when first introducing a concept. However, the defaults can be overridden to carry out sophisticated and targeted analyses. We will see in the Case Studies below how this concise syntax encourages sound analysis among students.

Pedagogy Case Studies

The following two case studies illustrate how grama is designed to support *pedagogy*: the formal method and practice of teaching. In particular, grama is designed for an active learning pedagogy [FEM+14], a style of teaching characterized by active student engagement.

Teachable Moments through Errors for Learners

An advantage of a unified modeling environment like grama is the opportunity to introduce design *errors for learners* in order to provide teachable moments.

It is common in probabilistic modeling to make problematic assumptions. For instance, Cullen and Frey [CF99] note that modelers frequently and erroneously treat the normal distribution as a default choice for all unknown quantities. Another common issue is to assume, by default, the independence of all random inputs to a model. This is often done *tacitly* – with the independence assumption unstated. These assumptions are problematic, as they can adversely impact the validity of a probabilistic analysis [dRFI21].

To highlight the dependency issue for novice modelers, grama uses error messages to provide just-in-time feedback to a user who does not articulate their modeling choices. For example, the following code builds a model with no dependency structure specified. The result is an error message that summarizes the conceptual issue and points the user to a primer on random variable modeling.

```
md_flawed = (
    gr.Model("An example model")
    >> gr.cp_vec_function(
```

```
        fun=lambda df: gr.df_make(f=df.x+df.y+df.z),
        var=["x", "y", "z"],
        out=["f"],
    )
    >> gr.cp_bounds(x=(-1, +1))
    >> gr.cp_marginals(
        y=gr.marg_mom("norm", mean=0, sd=1),
        z=gr.marg_mom("uniform", mean=0, sd=1),
    )
    ## NOTE: No dependency specified
)
(
    md_flawed
    ## This code will throw an Error
    >> gr.ev_sample(n=1000, df_det="nom")
)
```

Error `ValueError`: Present model copula must be defined for sampling. Use `CopulaIndependence` only when inputs can be guaranteed independent. See the Documentation chapter on Random Variable Modeling for more information. https://py-grama.readthedocs.io/en/latest/source/rv_modeling.html

Grama is designed both as a teaching tool and a scientific modeling toolkit. For the student, grama offers teachable moments to help the novice grow as a modeler. For the scientist, grama enforces practices that promote scientific reproducibility.

Encouraging Sound Analysis

As mentioned above, concise grama syntax is desirable to *encourage sound analysis practices*. Grama is designed to support higher-level learning outcomes [Blo56]. For instance, rather than focusing on *applying* programming constructs to generate model results, grama is intended to help users *study* model results ("evaluate," according to Bloom's Taxonomy). Sound computational analysis demands study of simulation results (e.g., to check for numerical instabilities). This case study makes this learning outcome distinction concrete by considering *parameter sweeps*.

Generating a parameter sweep similar to Figure 2 with standard Python libraries requires a considerable amount of boilerplate code, manual coordination of model information, and explicit loop construction. The following code generates parameter sweep data using standard libraries. Note that this code sweeps through values of `x` holding values of `y` fixed; additional code would be necessary to construct a sweep through `y`[2].

```
## Parameter sweep: Manual approach
# Gather model info
x_lo = -1; x_up = +1;
y_lo = -1; y_up = +1;
f_model = lambda x, y: x**2 * y
# Analysis parameters
nx = 10                 # Grid resolution for x
y_const = [-1, 0, +1] # Constant values for y
# Generate data
data = np.zeros((nx * len(y_const), 3))
for i, x in enumerate(
        np.linspace(x_lo, x_up, num=nx)
    ):
    for j, y in enumerate(y_const):
        data[i + j*nx, 0] = f_model(x, y)
        data[i + j*nx, 1] = x
        data[i + j*nx, 2] = y
# Package data for visual
df_manual = pd.DataFrame(
    data=data,
    columns=["f", "x", "y"],
)
```

2. Code assumes `import numpy as np; import pandas as pd`.

The ability to write low-level programming constructs – such as the loops above – is an obviously worthy learning outcome in a course on scientific computing. However, not all courses should focus on low-level programming constructs. Grama is not designed to support low-level learning outcomes; instead, the package is designed to support a "coding to learn" philosophy [Bar16] focused on higher-order learning outcomes to support sound modeling practices.

Parameter sweep functionality can be achieved in grama without explicit loop management and with sensible defaults for the analysis parameters. This provides a "quick and dirty" tool to inspect a model's behavior. A grama approach to parameter sweeps is shown below.

```
## Parameter sweep: Grama approach
# Gather model info
md_gr = (
    gr.Model()
    >> gr.cp_vec_function(
        fun=lambda df: gr.df_make(f=df.x**2 * df.y),
        var=["x", "y"],
        out=["f"],
    )
    >> gr.cp_bounds(
        x=(-1, +1),
        y=(-1, +1),
    )
)
# Generate data
df_gr = gr.eval_sinews(
    md_gr,
    df_det="swp",
    n_sweeps=3,
)
```

Once a model is implemented in grama, generating and visualizing a parameter sweep is trivial, requiring just two lines of code and zero initial choices for analysis parameters. The practical outcome of this software design is that users will tend to *self-initiate* parameter sweeps: While students will rarely choose to write the extensive boilerplate code necessary for a parameter sweep (unless required to do so), students writing code in grama will tend to self-initiate sound analysis practices.

For example, the following code is unmodified from a student report[3]. The original author implemented an ordinary differential equation model to simulate the track time `"finish_time"` of an electric formula car, and sought to study the impact of variables such as the gear ratio `"GR"` on `"finish_time"`. While the assignment did not require a parameter sweep, the student chose to carry out their own study. The code below is a self-initiated parameter sweep of the track time model.

```
## Unedited student code
md_car = (
    gr.Model("Accel Model")
    >> gr.cp_function(
        fun = calculate_finish_time,
        var = ["GR", "dt_mass", "I_net" ],
        out = ["finish_time"],
    )

    >> gr.cp_bounds(
        GR=(+1,+4),
        dt_mass=(+5,+15),
        I_net=(+.2,+.3),
```

3. Included with permission of the author, on condition of anonymity.

```
    )
)

gr.plot_auto(
    gr.eval_sinews(
        md_car,
        df_det="swp",
        #skip=True,
        n_density=20,
        n_sweeps=5,
        seed=101,
    )
)
```

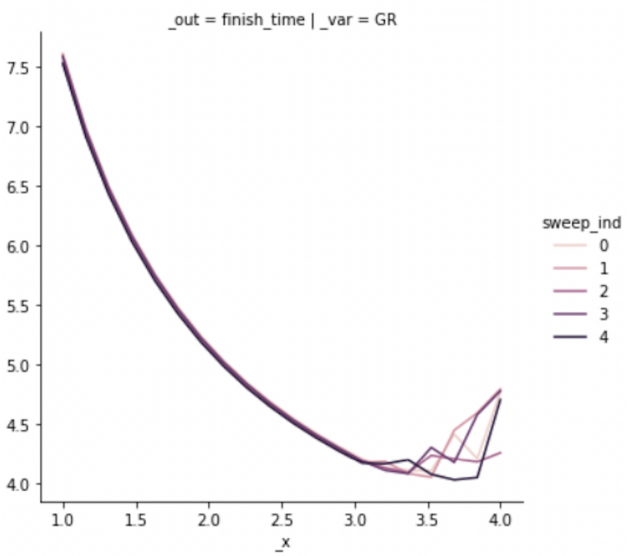


Fig. 3: *Input sweep generated from the student code above. The image has been cropped for space, and the results are generated with an older version of grama. The jagged response at higher values of the input are evidence of solver instabilities.*

The parameter sweep shown in Figure 2 gives an overall impression of the effect of input `"GR"` on the output `"finish_time"`. This particular input tends to dominate the results. However, variable results at higher values of `"GR"` provide evidence of numerical instability in the ODE solver underlying the model. Without this sort of model evaluation, the student author would not have discovered the limitations of the model.

Exploratory Model Analysis Case Study

This final case study illustrates how grama supports exploratory model analysis. This iterative process is a computational approach to mining insights into physical systems. The following use case illustrates the approach by considering the design of boat hull cross-sections.

Static Stability of Boat Hulls

Stability is a key consideration in boat hull design. One of the most fundamental aspects of stability is *static stability*; the behavior of a boat when perturbed away from static equilibrium [LE00]. Figure 4 illustrates the physical mechanism governing stability at small perturbations from an upright orientation.

As a boat is rotated away from its upright orientation, its center of buoyancy (COB) will tend to migrate. If the boat is in vertical equilibrium, its buoyant force will be equal in magnitude to its weight. A stable boat is a hull whose COB migrates in such a way

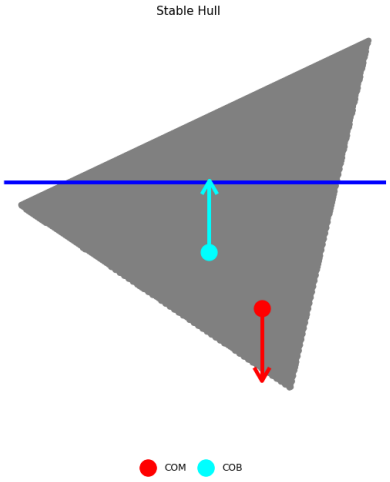


Fig. 4: *Schematic boat hull rotated to* 22.5°. *The forces due to gravity and buoyancy act at the center of mass (COM) and center of buoyancy (COB), respectively. Note that this hull is upright stable, as the couple will rotate the boat to upright.*

that a restoring torque is generated (Fig. 4). However, this upright stability is not guaranteed; Figure 5 illustrates a boat design that does not provide a restoring torque near its upright angle. An upright-unstable boat will tend to capsize spontaneously.

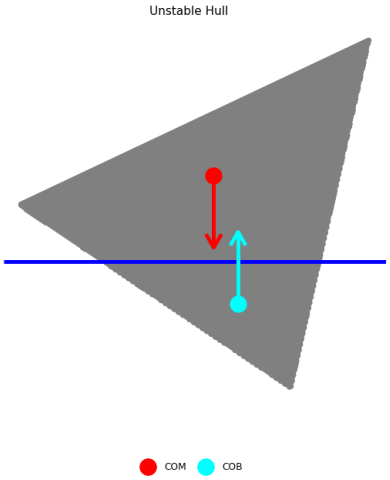


Fig. 5: *Schematic boat hull rotated to* 22.5°. *Gravity and buoyancy are annotated as in Figure 4. Note that this hull is upright unstable, as the couple will rotate the boat away from upright.*

Naval engineers analyze the stability of a boat design by constructing a *moment curve*, such as the one pictured in Figure 6. This curve depicts the net moment due to buoyancy at various angles, assuming the vessel is in vertical equilibrium. From this figure we can see that the design is upright-stable, as it possesses a negative slope at upright $\theta = 0°$. Note that a boat may not have an unlimited range of stability as Figure 6 exhibits an *angle of vanishing stability* (AVS) beyond which the boat does not recover to upright.

The classical way to build intuition about boat stability is via mathematical derivations [LE00]. In the following section we present an alternative way to build intuition through exploratory

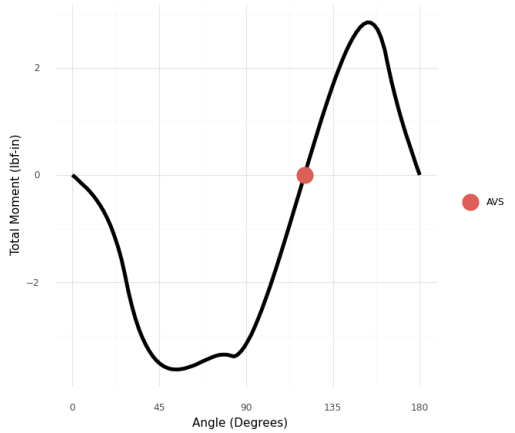


Fig. 6: *Total moment on a boat hull as it is rotated through* 180°. *A negative slope at upright* $\theta = 0°$ *is required for upright stability. Stability is lost at the* angle of vanishing stability *(AVS).*

model analysis.

EMA for Insight Mining

Generation and post-processing of the moment curve are implemented in the grama model `md_performance`[4]. This model parameterizes a 2d boat hull via its height `H`, width `W`, shape of corner `n`, the vertical height of the center of mass `f_com` (as a fraction of the height), and the *displacement ratio* `d` (the ratio of the boat's mass to maximum water mass displaced). Note that a boat with `d > 1` is incapable of flotation. A smaller value of `d` corresponds to a boat that floats higher in the water. The model `md_performance` returns `stability = -dMdtheta_0` (the negative of the moment curve slope at upright) as well as the `mass` and AVS `angle`. A positive value of `stability` indicates upright stability, while a larger value of `angle` indicates a wider range of stability.

The EMA process begins by generating data from the model. However, the generation of a moment curve is a nontrivial calculation. One should exercise care in choosing an initial sample of designs to analyze. The statistical problem of selecting efficient input values for a computer model is called the *design of computer experiments* [SSW89]. The grama verb *gr.tf_sp()* implements the support points algorithm [MJ18] to reduce a large dataset of target points to a smaller (but representative) sample. The following code generates a sample of input design values via `gr.ev_sample()` with the `skip=True` argument, uses `gr.tf_sp()` to "compact" this large sample, then evaluates the performance model at the smaller sample.

```
df_boats = (
    md_performance
    >> gr.ev_sample(
        n=5e3,
        df_det="nom",
        seed=101,
        skip=True,
    )
    >> gr.tf_sp(n=1000, seed=101)
    >> gr.tf_md(md=md_performance)
)
```

With an initial sample generated, we can perform an exploratory analysis relating the inputs and outputs. The verb `gr.tf_iocorr()` computes correlations between every pair of input variables `var` and outputs `out`. The routine also attaches metadata, enabling an autoplot as a tileplot of the correlation values.

```
(
    df_boats
    >> gr.tf_iocorr(
        var=["H", "W", "n", "d", "f_com"],
        out=["mass", "angle", "stability"],
    )
    >> gr.pt_auto()
)
```

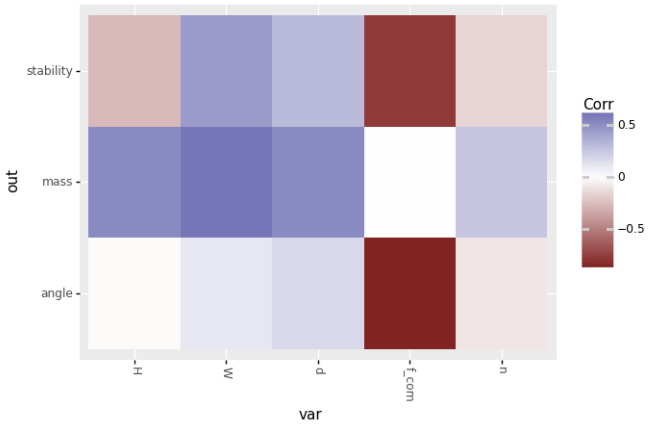


Fig. 7: *Tile plot of input/output correlations; autoplot gr.pt_auto() visualization of gr.tf_iocorr() output.*

The correlations in Figure 7 suggest that `stability` is positively impacted by increasing the width `W` and displacement ratio `d` of a boat, and by decreasing the height `H`, shape factor `n`, and vertical location of the center of mass `f_com`. The correlations also suggest a similar impact of each variable on the AVS `angle`, but with a weaker dependence on `H`. These results also suggest that `f_com` has the strongest effect on both `stability` and `angle`.

Correlations are a reasonable first-check of input/output behavior, but linear correlation quantifies only an average, linear association. A second-pass at the data would be to fit an accurate surrogate model and inspect parameter sweeps. The following code defines a gaussian process fit [RW05] for both `stability` and `angle`, and estimates model error using k-folds cross validation [JWHT13]. Note that a non-default kernel is necessary for a reasonable fit of the latter output[5].

```
## Define fitting procedure
ft_common = gr.ft_gp(
    var=["H", "W", "n", "d", "f_com"],
    out=["angle", "stability"],
    kernels=dict(
        stability=None,  # Use default
        angle=RBF(length_scale=0.1),
    )
)
## Estimate model accuracy via k-folds CV
(
    df_boats
    >> gr.tf_kfolds(
        ft=ft_common,
        out=["angle", "stability"],
    )
)
```

4. The analysis reported here is available as a jupyter notebook.

5. `RBF` is imported as `from sklearn.gaussian_process.kernels import RBF`.

angle	stability	k
0.771	0.979	0
0.815	0.976	1
0.835	0.95	2
0.795	0.962	3
0.735	0.968	4

TABLE 1: *Accuracy (R^2) estimated via k-fold cross validation of gaussian process model.*

The k-folds CV results (Tab. 1) suggest a highly accurate model for `stability`, and a moderately accurate model for `angle`. The following code defines the surrogate model over a domain that includes the original dataset, and performs parameter sweeps across all inputs.

```
md_fit = (
    df_boats
    >> ft_common()
    >> gr.cp_marginals(
        H=gr.marg_mom("uniform", mean=2.0, cov=0.30),
        W=gr.marg_mom("uniform", mean=2.5, cov=0.35),
        n=gr.marg_mom("uniform", mean=1.0, cov=0.30),
        d=gr.marg_mom("uniform", mean=0.5, cov=0.30),
        f_com=gr.marg_mom(
            "uniform",
            mean=0.55,
            cov=0.47,
        ),
    )
    >> gr.cp_copula_independence()
)

(
    md_fit
    >> gr.ev_sinews(df_det="swp", n_sweeps=5)
    >> gr.pt_auto()
)
```

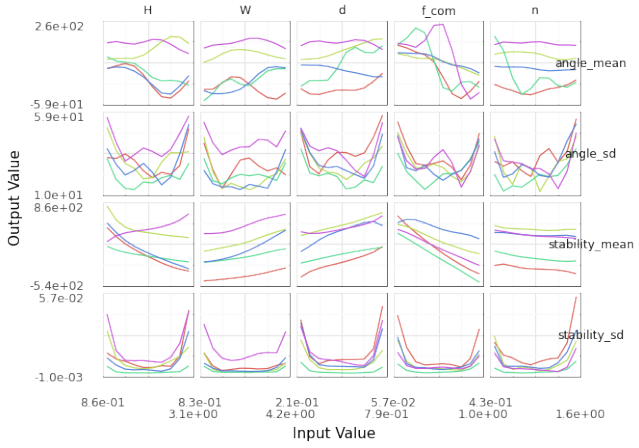


Fig. 8: *Parameter sweeps for fitted GP model. Model `*_mean` and predictive uncertainty `*_sd` values are reported for each output `angle`, `stability`.*

Figure 8 displays parameter sweeps for the surrogate model of `stability` and `angle`. Note that the surrogate model reports both a mean trend `*_mean` and a predictive uncertainty `*_sd`. The former is the model's prediction for future values, while the latter quantifies the model's confidence in each prediction.

The parameter sweeps of Figure 8 show a consistent and strong effect of `f_com` on the `stability_mean` of the boat; note that all the sweeps across `f_com` for `stability_mean` tend to be monotone with a fairly steep slope. This is in agreement with the correlation results of Figure 7; the `f_com` sweeps tend to have the steepest slopes. Given the high accuracy of the model for `stability` (as measured by k-folds CV), this trend is reasonably trustworthy.

Direction	H	W	n	d	f_com
1	-0.0277	0.0394	-0.1187	0.4009	-0.9071
2	-0.6535	0.3798	-0.0157	-0.6120	-0.2320

TABLE 2: *Subspace weights in `df_weights`.*

However, the same figure shows an inconsistent (non-monotone) effect of most inputs on the AVS `angle_mean`. These results are in agreement with the k-fold CV results shown above. Clearly, the surrogate model is untrustworthy, and we should resist trusting conclusions from the parameter sweeps for `angle_mean`. This undermines the conclusion we drew from the input/output correlations pictured in Figure 7. Clearly, `angle` exhibits more complex behavior than a simple linear correlation with each of the boat design variables.

A different analysis of the boat hull `angle` data helps develop useful insights. We pursue an active subspace analysis of the data to reduce the dimensionality of the input space by identifying directions that best explain variation in the output [dCI17], [Con15]. The verb `gr.tf_polyridge()` implements the variable projection algorithm of Hokanson and Constantine [HC18]. The following code pursues a two-dimensional reduction of the input space. Note that the hyperparameter `n_degree=6` is set via a cross-validation study.

```
## Find two important directions
df_weights = (
    df_boats
    >> gr.tf_polyridge(
        var=["H", "W", "n", "d", "f_com"],
        out="angle",
        n_degree=6,  # Set via CV study
        n_dim=2,     # Seek 2d subspace
    )
)
```

The subspace weights are reported in Table 2. Note that the leading direction `1` is dominated by the displacement ratio `d` and COM location `f_com`. Essentially, this describes the "loading" of the vessel. The second direction corresponds to "widening and shortening" of the hull cross-section (in addition to lowering `d` and `f_com`).

Using the subspace weights in Table 2 to produce a 2d projection of the feature space enables visualizing all boat geometries in a single plot. Figure 9 reveals that this 2d projection is very successful at separating universally-stable (`angle==180`), upright-unstable (`angle==0`), and intermediate cases (`0 < angle < 180`). Intermediate cases are concentrated at higher values of the second active variable. There is a phase transition between universally-stable and upright-unstable vessels at lower values of the second active variable.

Interpreting Figure 9 in light of Table 2 provides us with deep insight about boat stability: Since active variable 1 corresponds to loading (high displacement ratio `d` with a low COM `f_com`), we can see that the boat's loading conditions are key to determining its stability. Since active variable 2 depends on the aspect ratio

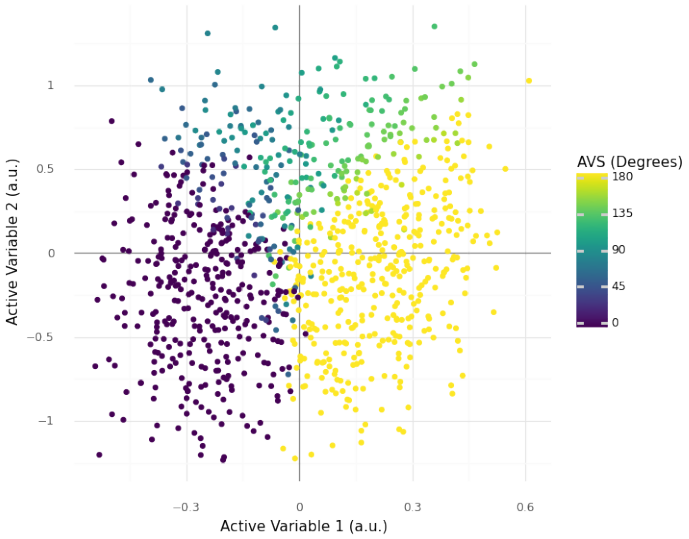


Fig. 9: *Boat design feature vectors projected to 2d active subspace. The origin corresponds to the mean feature vector.*

(higher width, shorter height), Figure 9 suggests that only wider boats will tend to exhibit intermediate stability.

Conclusions

Grama is a Python implementation of a grammar of model analysis. The grammar's design supports an active learning approach to teaching sound scientific modeling practices. Two case studies demonstrated the teaching benefits of grama: *errors for learners* help guide novices toward a more sound analysis, while concise syntax encourages novices to carry out sound analysis practices. Grama can also be used for exploratory model analysis (EMA) – an exploratory procedure to mine a scientific model for useful insights. A case study of boat hull design demonstrated EMA. In particular, the example explored and explained the relationship between boat design parameters and two metrics of boat stability.

Several ideas from the grama project are of interest to other practitioners and developers in scientific computing. Grama was designed to support model analysis *under uncertainty*. However, the **data/model and four-verb ontology** (Fig. 1) underpinning grama is a much more general idea. This design enables very concise model analysis syntax, which provides much of the benefit behind grama.

The design idiom of **errors for learners** is not simply focused on writing "useful" error messages, but is rather a design orientation to use errors to introduce teachable moments. In addition to writing error messages "for humans" [Bry20], an *errors for learners* philosophy designs errors not simply to avoid fatal program behavior, but rather introduces exceptions to prevent conceptually invalid analyses. For instance, in the case study presented above, designing *gr.tf_sample()* to assume independent random inputs when a copula is unspecified would lead to code that throws errors less frequently. However, this would silently endorse the conceptually problematic mentality of "independence is the default." While throwing an error message for an unspecified dependence structure leads to more frequent errors, it serves as a frequent reminder that dependency is an important part of a model involving random inputs.

Finally, exploratory model analysis holds benefits for both learners and practitioners of scientific modeling. EMA is an alternative to derivation for the activities in an active learning approach. Rather than structuring courses around deriving and implementing scientific models, course exercises could have students explore the behavior of a pre-implemented model to better understand physical phenomena. Lorena Barba [Bar16] describes some of the benefits in this style of lesson design. EMA is also an important part of the modeling practitioner's toolkit as a means to verify a model's implementation and to develop new insights. Grama supports both novices and practitioners in performing EMA through a concise syntax.

References

[AFD+21] Riya Aggarwal, Mira Flynn, Sam Daitzman, Diane Lam, and Zachary Riggins del Rosario. A qualitative study of engineering students' reasoning about statistical variability. In *2021 Fall ASEE Middle Atlantic Section Meeting*, 2021. URL: https://peer.asee.org/38421.

[Bar16] Lorena Barba. Computational thinking: I do not think it means what you think it means. Technical report, 2016. URL: https://lorenabarba.com/blog/computational-thinking-i-do-not-think-it-means-what-you-think-it-means/.

[Bie20] Przemyslaw Biecek. *ceterisParibus: Ceteris Paribus Profiles*, 2020. R package version 0.4.2. URL: https://cran.r-project.org/package=ceterisParibus.

[Blo56] Benjamin Samuel Bloom. *Taxonomy of educational objectives: The classification of educational goals*. Addison-Wesley Longman Ltd., 1956.

[Bry20] Jennifer Bryan. object of type closure is not subsettable. 2020. rstudio::conf 2020. URL: https://rstd.io/debugging.

[BTSA12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012. URL: https://ieeexplore.ieee.org/document/6468504, doi:10.1109/SC.2012.71.

[CF99] Alison C Cullen and H Christopher Frey. *Probabilistic Techniques In Exposure Assessment: A Handbook For Dealing With Variability And Uncertainty In Models And Inputs*. Springer Science & Business Media, 1999.

[Con15] Paul G. Constantine. *Active Subspaces: Emerging Ideas for Dimension Reduction in Parameter Studies*. SIAM Philadelphia, 2015. doi:10.1137/1.9781611973860.

[dCI17] Zachary del Rosario, Paul G. Constantine, and Gianluca Iaccarino. Developing design insight through active subspaces. In *19th AIAA Non-Deterministic Approaches Conference*, page 1090, 2017. URL: https://arc.aiaa.org/doi/10.2514/6.2017-1090, doi:10.2514/6.2017-1090.

[dR20] Zachary del Rosario. Grama: A grammar of model analysis. *Journal of Open Source Software*, 5(51):2462, 2020. URL: https://doi.org/10.21105/joss.02462, doi:10.21105/joss.02462.

[dRFI21] Zachary del Rosario, Richard W Fenrich, and Gianluca Iaccarino. When are allowables conservative? *AIAA Journal*, 59(5):1760–1772, 2021. URL: https://doi.org/10.2514/1.J059578, doi:10.2514/1.J059578.

[EVB+20] M Esmaily, L Villafane, AJ Banko, G Iaccarino, JK Eaton, and A Mani. A benchmark for particle-laden turbulent duct flow: A joint computational and experimental study. *International Journal of Multiphase Flow*, 132:103410, 2020. URL: https://www.sciencedirect.com/science/article/abs/pii/S030193222030519X, doi:10.1016/j.ijmultiphaseflow.2020.103410.

[FEM+14] Scott Freeman, Sarah L Eddy, Miles McDonough, Michelle K Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences*, 111(23):8410–8415, 2014. doi:10.1073/pnas.1319030111.

[HC18] Jeffrey M Hokanson and Paul G Constantine. Data-driven polynomial ridge approximation using variable projection. *SIAM Journal on Scientific Computing*, 40(3):A1566–A1589, 2018. doi:10.1137/17M1117690.

[HK21] Jan Katins gdowding austin matthias-k Tyler Funnell Florian Finkernagel Jonas Arnfred Dan Blanchard et al. Hassan Kibirige, Greg Lamp. has2k1/plotnine: v0.8.0. Mar 2021. doi:10.5281/zenodo.4636791.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*, volume 112. Springer, 2013. URL: https://www.statlearning.com/.

[KBB19] Michał Kuźba, Ewa Baranowska, and Przemysław Biecek. pyceterisparibus: explaining machine learning models with ceteris paribus profiles in python. *Journal of Open Source Software*, 4(37):1389, 2019. URL: https://joss.theoj.org/papers/10.21105/joss.01389, doi:10.21105/joss.01389.

[KN05] Andy Keane and Prasanth Nair. *Computational Approaches For Aerospace Design: The Pursuit Of Excellence*. John Wiley & Sons, 2005.

[KSS21] Daniel Kahneman, Olivier Sibony, and Cass R Sunstein. *Noise: A flaw in human judgment*. Little, Brown, 2021.

[LE00] Lars Larsson and Rolf Eliasson. *Principles of Yacht Design*. McGraw Hill Companies, 2000.

[MJ18] Simon Mak and V Roshan Joseph. Support points. *The Annals of Statistics*, 46(6A):2562–2592, 2018. doi:10.1214/17-AOS1629.

[MTW+22] Kazuki Maeda, Thiago Teixeira, Jonathan M Wang, Jeffrey M Hokanson, Caetano Melone, Mario Di Renzo, Steve Jones, Javier Urzay, and Gianluca Iaccarino. An integrated heterogeneous computing framework for ensemble simulations of laser-induced ignition. *arXiv preprint arXiv:2202.02319*, 2022. URL: https://arxiv.org/abs/2202.02319, doi:10.48550/arXiv.2202.02319.

[MV15] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015. URL: https://dl.acm.org/doi/10.1145/2788396, doi:10.1145/2788396.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[RW05] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 11 2005. URL: https://doi.org/10.7551/mitpress/3206.001.0001, doi:10.7551/mitpress/3206.001.0001.

[SKA+14] Jeffrey P Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, and Dimitri J Mavriplis. Cfd vision 2030 study: A path to revolutionary computational aerosciences. Technical report, 2014. URL: https://ntrs.nasa.gov/citations/20140003093.

[SS13] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6567202, doi:10.1109/CTS.2013.6567202.

[SSW89] Jerome Sacks, Susannah B. Schiller, and William J. Welch. Designs for computer experiments. *Technometrics*, 31(1):41–47, 1989. URL: http://www.jstor.org/stable/1270363, doi:10.2307/1270363.

[WAB+19] Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, et al. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. doi:10.21105/joss.01686.

[Wic14] Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014. doi:10.18637/jss.v059.i10.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a.

Low Level Feature Extraction for Cilia Segmentation

Meekail Zain[‡†*], Eric Miller[§†], Shannon P Quinn[‡¶], Cecilia Lo[‖]

Abstract—Cilia are organelles found on the surface of some cells in the human body that sweep rhythmically to transport substances. Dysfunction of ciliary motion is often indicative of diseases known as ciliopathies, which disrupt the functionality of macroscopic structures within the lungs, kidneys and other organs [LWL+18]. Phenotyping ciliary motion is an essential step towards understanding ciliopathies; however, this is generally an expert-intensive process [QZD+15]. A means of automatically parsing recordings of cilia to determine useful information would greatly reduce the amount of expert intervention required. This would not only improve overall throughput, but also mitigate human error, and greatly improve the accessibility of cilia-based insights. Such automation is difficult to achieve due to the noisy, partially occluded and potentially out-of-phase imagery used to represent cilia, as well as the fact that cilia occupy a minority of any given image. Segmentation of cilia mitigates these issues, and is thus a critical step in enabling a powerful pipeline. However, cilia are notoriously difficult to properly segment in most imagery, imposing a bottleneck on the pipeline. Experimentation on and evaluation of alternative methods for feature extraction of cilia imagery hence provide the building blocks of a more potent segmentation model. Current experiments show up to a 10% improvement over base segmentation models using a novel combination of feature extractors.

Index Terms—cilia, segmentation, u-net, deep learning

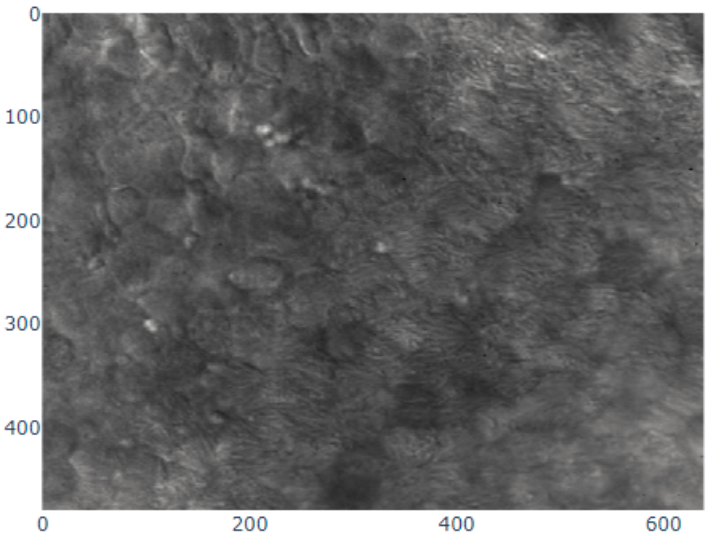


Fig. 1: *A sample frame from the cilia dataset*

Introduction

Cilia are organelles found on the surface of some cells in the human body that sweep rhythmically to transport substances [Ish17]. Dysfunction of ciliary motion often indicates diseases known as ciliopathies, which on a larger scale disrupt the functionality of structures within the lungs, kidneys and other organs. Phenotyping ciliary motion is an essential step towards understanding ciliopathies. However, this is generally an expert-intensive process [LWL+18], [QZD+15]. A means of automatically parsing recordings of cilia to determine useful information would greatly reduce the amount of expert intervention required, thus increasing throughput while alleviating the potential for human error. Hence, Zain et al. (2020) discuss the construction of a generative pipeline to model and analyze ciliary motion, a prevalent field of investigation in the Quinn Research Group at the University of Georgia [ZRS+20].

The current pipeline consists of three major stages: preprocessing, where segmentation masks and optical flow representations are created to supplement raw cilia video data; appearance, where a model learns a condensed spacial representation of the cilia; and dynamics, which learns a representation from the video, encoded as a series of latent points from the appearance module. In the primary module, the segmentation mask is essential in scoping downstream analysis to the cilia themselves, so inaccuracies at this stage directly affect the overall performance of the pipeline. However, due to the high variance of ciliary structure, as well as the noisy and out-of-phase imagery available, segmentation attempts have been prone to error.

While segmentation masks for such a pipeline could be manually generated, the process requires intensive expert labor [DvBB+21]. Requiring manual segmentation before analysis thus greatly increases the barrier to entry for this tool. Not only would it increase the financial strain of adopting ciliary analysis as a clinical tool, but it would also serve as an insurmountable barrier to entry for communities that do not have reliable access to such clinicians in the first place, such as many developing nations and rural populations. Not only can automated segmentation mitigate these barriers to entry, but it can also simplify existing treatment and analysis infrastructure. In particular, it has the potential to reduce the magnitude of work required by an expert clinician, thereby

† *These authors contributed equally.*
∗ *Corresponding author: meekail.zain@uga.edu*
‡ *Department of Computer Science, University of Georgia, Athens, GA 30602 USA*
§ *Institute for Artificial Intelligence, University of Georgia, Athens, GA 30602 USA*
¶ *Department of Cellular Biology, University of Georgia, Athens, GA 30602 USA*
‖ *Department of Developmental Biology, University of Pittsburgh, Pittsburgh, PA 15261 USA*

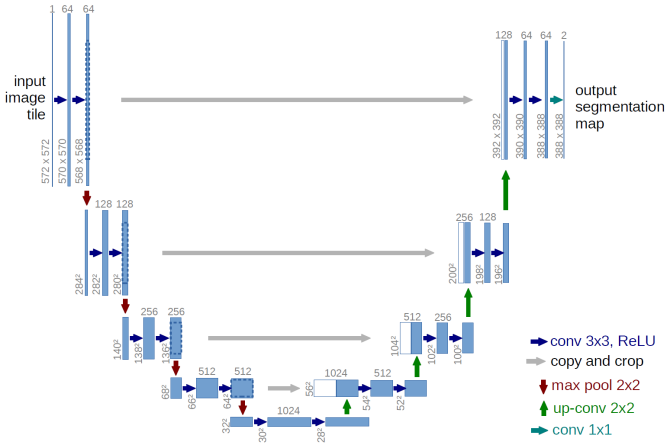


Fig. 2: *The classical U-Net architecture, which serves as both a baseline and backbone model for this research*

decreasing costs and increasing clinician throughput [QZD+15], [ZRS+20]. Furthermore, manual segmentation imparts clinician-specific bias which reduces the reproducability of results, making it difficult to verify novel techniques and claims [DvBB+21].

A thorough review of previous segmentation models, specifically those using the same dataset, shows that current results are poor, impeding tasks further along the pipeline. For this study, model architectures utilize various methods of feature extraction that are hypothesized to improve the accuracy of a base segmentation model, such as using zero-phased PCA maps and Sparse Autoencoder reconstructions with various parameters as a data augmentation tool. Various experiments with these methods provide a summary of both qualitative and quantitative results necessary in ascertaining the viability for such feature extractors to aid in segmentation.

Related Works

Lu et. al. (2018) utilized a Dense Net segmentation model as an upstream to a CNN-based Long Short-Term Memory (LSTM) time-series model for classifying cilia based on spatiotemporal patterns [LMZ+18]. While the model reports good classification accuracy and a high F-1 score, the underlying dataset only contains 75 distinct samples and the results must therefore be taken with great care. Furthermore, Lu et. al. did not report the separate performance of the upstream segmentation network. Their approach did, however, inspire the follow-up methodology of Zain et. al. (2020) for segmentation. In particular, they employ a Dense Net segmentation model as well, however they first augment the underlying images with the calculated optical flow. In this way, their segmentation strategy employs both spatial *and* temporal information. To compare against [LMZ+18], the authors evaluated their segmentation model in the same way—as an upstream to an CNN/LSTM classification network. Their model improved the classification accuracy two points above that of Charles et. al. (2018). Their reported intersection-over-union (IoU) score is 33.06% and marks the highest performance achieved on this dataset.

One alternative segmentation model, often used in biomedical image processing and analysis, where labelled data sets are relatively small, is the U-Net architecture (2) [RFB15]. Developed by Ronneberger et. al., U-Nets consist of two parts: contraction and expansion. The contraction path follows the standard strategy of most convolutional neural networks (CNNs), where convolutions are followed by Rectified Linear Unit (ReLU) activation functions and max pooling layers. While max pooling downsamples the images, the convolutions double the number of channels. Upon expansion, up-convolutions are applied to up-sample the image while reducing the number of channels. At each stage, the network concatenates the up-sampled image with the image of corresponding size (cropped to account for border pixels) from a layer in the contracting path. A final layer uses pixel-wise (1×1) convolutions to map each pixel to a corresponding class, building a segmentation. Before training, data is generally augmented to provide both invariance in rotation and scale as well as a larger amount of training data. In general, U-Nets have shown high performance on biomedical data sets with low quantities of labelled images, as well as reasonably fast training times on graphics processing units (GPUs) [RFB15]. However, in a few past experiments with cilia data, the U-Net architecture has had low segmentation accuracy [LMZ+18]. Difficulties modeling cilia with CNN-based architectures include their fine high-variance structure, spatial sparsity, color homogeneity (with respect to the background and ambient cells), as well as inconsistent shape and distribution across samples. Hence, various enhancements to the pure U-Net model are necessary for reliable cilia segmentation.

Methodology

The U-Net architecture is the backbone of the model due to its well-established performance in the biomedical image analysis domain. This paper focuses on extracting and highlighting the underlying features in the image through various means. Therefore, optimization of the U-Net backbone itself is not a major consideration of this project. Indeed, the relative performance of the various modified U-Nets sufficiently communicates the efficacy of the underlying methods. Each feature extraction method will map the underlying raw image to a corresponding feature map. To evaluate the usefulness of these feature maps, the model concatenates these augmentations to the original image and use the aggregate data as input to a U-Net that is slightly modified to accept multiple input channels.

The feature extractors of interest are Zero-phase PCA sphering (ZCA) and a Sparse Autoencoder (SAE), on both of which the following subsections provide more detail. Roughly speaking, these are both lossy, non-bijective transformations which map a single image to a single feature map. In the case of ZCA, empirically the feature maps tend to preserve edges and reduce the rest of the image to arbitrary noise, thereby emphasizing local structure (since cell structure tends not to be well-preserved). The SAE instead acts as a harsh compression and filters out both linear and non-linear features, preserving global structure. Each extractor is evaluated by considering the performance of a U-Net model trained on multi-channel inputs, where the first channel is the original image, and the second and/or third channels are the feature maps extracted by these methods. In particular, the objective is for the doubly-augmented data, or the "composite" model, to achieve state-of-the-art performance on this challenging dataset.

The ZCA implementation utilizes SciPy linear algebra solvers, and both U-Net and SAE architectures use the PyTorch deep learning library. Next, the evaluation stage employs canonical segmentation quality metrics, such as the Jaccard score and Dice coefficient, on various models. When applied to the composite

model, these metrics determine any potential improvements to the state-of-the-art for cilia segmentation.

Cilia Data

As in the Zain paper, the input data is a limited set of grayscale cilia imagery, from both healthy patients and those diagnosed with ciliopathies, with corresponding ground truth masks provided by experts. The images are cropped to 128×128 patches. The images are cropped at random coordinates in order to increase the size and variance of the sample space, and each image is cropped a number of times proportional its resolution. Additionally, crops that contain less than fifteen percent cilia are excluded from the training/test sets. This method increases the size of the training set from 253 images to 1409 images. Finally, standard minmax contrast normalization maps the luminosity to the interval $[0,1]$.

Zero-phase PCA sphering (ZCA)

The first augmentation of the underlying data concatenates the input to the backbone U-Net model with the ZCA-transformed data. ZCA maps the underlying data to a version of the data that is "rotated" through the dataspace to ensure certain spectral properties. ZCA in effect can implicitly normalize the data using the most significant (by empirical variance) spatial features present across the dataset. Given a matrix X with rows representing samples and columns for each feature, a sphering (or whitening) transformation W is one which decorrelates X. That is, the covariance of WX must be equal to the identity matrix. By the spectral theorem, the symmetric matrix XX^T—the covariance matrix corresponding to the data, assuming the data is centered—can be decomposed into PDP^T, where P is an orthogonal matrix of eigenvectors and D a diagonal matrix of corresponding eigenvalues of the covariance matrix. ZCA uses the sphering matrix $W = PD^{-1/2}P^T$ and can be thought of as a transformation into the eigenspace of its covariance matrix—projection onto the data's principal axes, as the minimal projection residual is onto the axes with maximal variance—followed by normalization of variance along every axis and rotation back into the original image space. In order to reduce the amount of two-way correlation in images, Krizhevsky applies ZCA whitening to preprocess CIFAR-10 data before classification and shows that this process nicely preserves features, such as edges [LjWD19].

This ZCA implementation uses the Python SciPy library (SciPy), which builds on top of low-level hardware-optimized routines such as BLAS and LAPACK to efficiently calculate many linear algebra operations. In particular, these expirements implement ZCA as a generalized whitening technique. While normal the normal ZCA calculation selects a whitening matrix $W = PD^{-\frac{1}{2}}P^T$, a more applicable alternative is $W = P\sqrt{(D+\varepsilon I)^{-1}}P^T$ where ε is a hyperparameter which attenuates eigenvalue sensitivity. This new "whitening" is actually not a proper whitening since it does not guarantee an identity covariance matrix. It does however serve a similar purpose and actually lends some benefits.

Most importantly, it is indeed a generalization of canonical ZCA. That is to say, $\varepsilon = 0$ recovers canonical ZCA, and $\lambda \to \sqrt{\frac{1}{\lambda}}$ provides the spectrum of W on the eigenvalues. Otherwise, $\varepsilon > 0$ results in the map $\lambda \to \sqrt{\frac{1}{\lambda+\varepsilon}}$. In this case, while *all* eigenvalues map to smaller values compared to the original map, the smallest eigenvalues map to significantly smaller values compared to the original map. This means that ε serves to "dampen" the effects of whitening for particularly small eigenvalues. This is a valuable feature since often times in image analysis low eigenvalues (and the span of their corresponding eigenvectors) tend to capture high-frequency data. Such data is essential for tasks such as texture analysis, and thus tuning the value of ε helps to preserve this data. ZCA maps for various values of ε on a sample image are shown in figure 3.

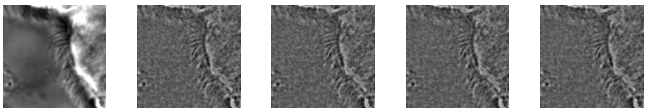


Fig. 3: *Comparison of ZCA maps on a cilia sample image with various levels of ε. The original image is followed by maps with $\varepsilon = 1e-4$, $\varepsilon = 1e-5$, $\varepsilon = 1e-6$, and $\varepsilon = 1e-7$, from left to right.*

Sparse Autoencoder (SAE)

Similar in aim to ZCA, an SAE can augment the underlying images to further filter and reduce noise while allowing the construction and retention of potentially nonlinear spatial features. Autoencoders are deep learning models that first compress data into a low-level latent space and then attempt to reconstruct images from the low-level representation. SAEs in particular add an additional constraint, usually via the loss function, that encourages sparsity (i.e., less activation) in hidden layers of the network. Xu et. al. use the SAE architecture for breast cancer nuclear detection and show that the architecture preserves essential, high-level, and often nonlinear aspects of the initial imagery—even when unlabelled—such as shape and color [XXL+16]. An adaptation of the first two terms of their loss function enforces sparsity:

$$\mathscr{L}_{SAE}(\theta) = \frac{1}{N}\sum_{k=1}^{N}(L(x(k), d_{\hat{\theta}}(e_{\check{\theta}}(x(k))))) + \alpha\frac{1}{n}\sum_{j=1}^{n} KL(\rho||\hat{\rho}).$$

The first term is a standard reconstruction loss (mean squared error), whereas the latter is the mean Kullback-Leibler (KL) divergence between $\hat{\rho}$, the activation of a neuron in the encoder, and ρ, the enforced activation. For the case of experiments performed here, $\rho = 0.05$ remains constant but values of α vary, specifically $1e-2$, $1e-3$, and $1e-4$, for each of which a static dataset is created for feeding into the segmentation model. Larger alpha prioritizes sparsity over reconstruction accuracy, which to an extent, is hypothesized to retain significant low-level features of the cilia. Reconstructions with various values of α are shown in figure 4

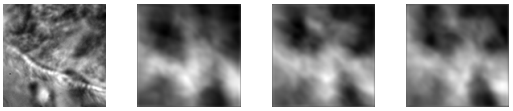
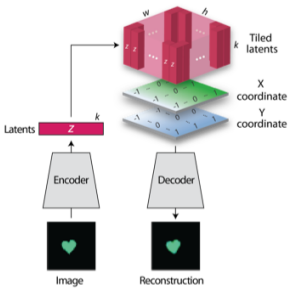


Fig. 4: *Comparison of SAE reconstructions from different training instances with various levels of α (the activation loss weight). From left to right: original image, $\alpha = 1e-2$ reconstruction, $\alpha = 1e-3$ reconstruction, $\alpha = 1e-4$ reconstruction.*

A significant amount of freedom can be found in potential architectural choices for SAE. A focus on low-medium complexity models both provides efficiency and minimizes overfitting and artifacts as consequence of degenerate autoencoding. One important danger to be aware of is that SAEs—and indeed, *all* AEs—are at risk of a degenerate solution wherein a sufficiently complex



Algorithm 1 Spatial Broadcast

Input: latents $\mathbf{z} \in \mathcal{R}^k$, width w, height h

Output: tiled latents $\mathbf{z}_{sb} \in \mathcal{R}^{h \times w \times (k+2)}$

$\mathbf{z}_b = \text{TILE}(\mathbf{z}, (h, w, 1))$

$\mathbf{x} = \text{LINSPACE}(-1, 1, w)$

$\mathbf{y} = \text{LINSPACE}(-1, 1, w)$

$\mathbf{x}_b, \mathbf{y}_b = \text{MESHGRID}(\mathbf{x}, \mathbf{y})$

$\mathbf{z}_{sb} = \text{CONCAT}([\mathbf{z}_b, \mathbf{x}_b, \mathbf{y}_b], \text{axis} = -1)$

return $\mathbf{z}_{sb}$

Fig. 5: *Illustration and pseudocode for Spatial Broadcast Decoding [WMBL19]*

decoder essentially learns to become a hashmap of arbitrary (and potentially random) encodings.

The SAE will therefore utilize a CNN architecture, as opposed to more modern transformer-style architectures, since the simplicity and induced spatial bias provide potent defenses against overfitting and mode collapse. Furthermore the encoder will use Spatial Broadcast Decoding (SBD) which provides a method for decoding from a latent vector using size-preserving convolutions, thereby preserving the spatial bias even in decoding, and eliminating the artifacts generated by alternate decoding strategies such as "transposed" convolutions [WMBL19].

Spatial Broadcast Decoding (SBD)

Spatial Broadcast Decoding provides an alternative method from "transposed" (or "skip") convolutions to upsample images in the decoder portion of CNN-based autoencoders. Rather than maintaining the square shape, and hence associated spatial properties, of the latent representation, the output of the encoder is reshaped into a single one-dimensional tensor per input image, which is then tiled to the shape of the desired image (in this case, 128×128). In this way, the initial dimension of the latent vector becomes the number of input channels when fed into the decoder, and two additional channels are added to represent 2-dimensional spatial coordinates. In its initial publication, SBD has been shown to provide effective results in disentangling latent space representations in various autoencoder models.

U-Net

All models use a standard U-Net and undergo the same training process to provide a solid basis for analysis. Besides the number of input channels to the initial model (1 plus the number of augmentation channels from SAE and ZCA, up to 3 total channels), the model architecture is identical for all runs. A single-channel (original image) U-Net first trains as a basis point for analysis. The model trains on two-channel inputs provided by ZCA (original image concatenated with the ZCA-mapped one) with various ε values for the dataset, and similarly SAE with various α values, train the model. Finally, composite models train with a few combinations of ZCA and SAE hyperparameters. Each training process uses binary cross entropy loss with a learning rate of $1e-3$ for 225 epochs.

Results

Figures 6, 7, 8, and 9 show masks produced on validation data from instances of the four model types. While the former three show results near the end of training (about 200-250 epochs),

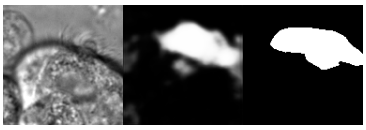


Fig. 6: *Artifacts generated during the training of U-Net. From left to right: original image, generated segmentation mask (pre-threshold), ground-truth segmentation mask*

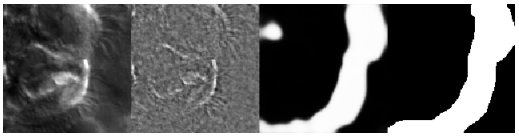


Fig. 7: *Artifacts generated during the training of ZCA+U-Net. From left to right: original image, ZCA-mapped image, generated segmentation mask (pre-threshold), ground-truth segmentation mask*

figure 9 was taken only 10 epochs into the training process. Notably, this model, the composite pipeline, produced usable artifacts in mere minutes of training, whereas other models did not produce similar results until after about 10-40 epochs.

Figure 10 provides a summary of experiments performed with *SAE* and *ZCA* augmented data, along with a few composite models and a base U-Net for comparison. These models were produced with data augmentation at various values of α (for the Sparse Autoencoder loss function) and ε (for ZCA) discussed above. While the table provides five metrics, those of primary importance are the Intersection over Union (IoU), or Jaccard Score, as well as the Dice (or F1) score, which are the most commonly used metrics for evaluating the performance of segmentation models. Most feature extraction models at least marginally improve the performance in of the U-Net in terms of IoU and Dice scores, and the best-performing composite model (with ε of $1e-4$ for ZCA and α of $1e-3$ for SAE) provide an improvement of approximately 10% from the base U-Net in these metrics. There does not seem to be an obvious correlation between which feature extraction hyperparameters provided the best performance for individual ZCA+U-Net and SAE+U-Net models versus those for the composite pipeline, but further experiments may assist in analyzing this possibility.

The base U-Net does outperform the others in precision,

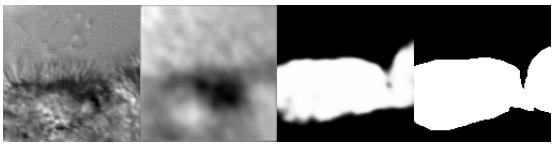


Fig. 8: *Artifacts generated during the training of SAE+U-Net. From left to right: original image, SAE-reconstructed image, generated segmentation mask (pre-threshold), ground-truth segmentation mask*

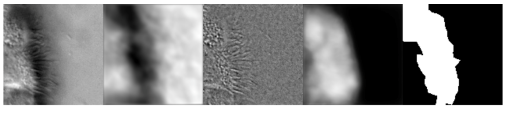


Fig. 9: *Artifacts generated 10 epochs into the training of the composite U-Net. From left to right: original image, ZCA-mapped image, SAE-mapped image, generated segmentation mask (pre-threshold), ground-truth segmentation mask*

Model	Extractor Parameters		Scores				
	ε (ZCA)	α (SAE)	IoU	Accuracy	Recall	Dice	Precision
U-Net (base)	—	—	0.399	0.759	0.501	0.529	0.692
ZCA + U-Net	$1e-4$	—	0.395	0.754	0.509	0.513	0.625
	$1e-5$	—	0.401	0.732	0.563	0.539	0.607
	$1e-6$	—	0.408	0.756	0.543	0.546	0.644
	$1e-7$	—	0.419	0.758	0.563	0.557	0.639
SAE + U-Net	—	$1e-2$	0.380	0.719	0.568	0.520	0.558
	—	$1e-3$	0.398	0.751	0.512	0.526	0.656
	—	$1e-4$	0.416	0.735	0.607	0.555	0.603
Composite	$1e-4$	$1e-2$	0.401	0.761	0.506	0.521	0.649
	$1e-4$	$1e-3$	**0.441**	0.767	0.580	**0.585**	0.661
	$1e-4$	$1e-4$	0.305	0.722	0.398	0.424	0.588
	$1e-5$	$1e-2$	0.392	0.707	**0.624**	0.530	0.534
	$1e-5$	$1e-3$	0.413	**0.770**	0.514	0.546	0.678
	$1e-5$	$1e-4$	0.413	0.751	0.565	0.550	0.619
	$1e-6$	$1e-2$	0.392	0.719	0.602	0.527	0.571
	$1e-6$	$1e-3$	0.395	0.759	0.480	0.521	**0.711**
	$1e-6$	$1e-4$	0.405	0.729	0.587	0.545	0.591
	$1e-7$	$1e-2$	0.383	0.753	0.487	0.503	0.655
	$1e-7$	$1e-3$	0.380	0.736	0.526	0.519	0.605
	$1e-7$	$1e-4$	0.293	0.674	0.445	0.418	0.487

Fig. 10: *A summary of segmentation scores on test data for a base U-Net model, ZCA+U-Net, SAE+U-Net, and a composite model, with various feature extraction hyperparameters. The best result for each scoring metric is in bold.*

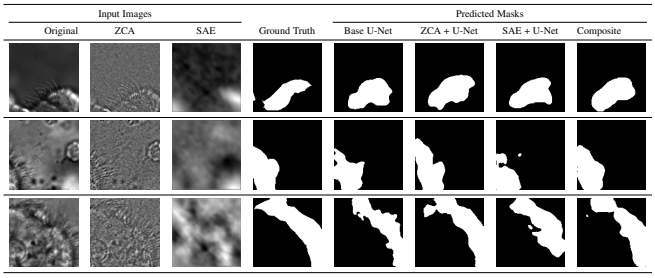


Fig. 11: *Comparison of predicted masks and ground truth for three test images. ZCA mapped images with $\varepsilon = 1e-4$ and SAE reconstructions with $\alpha = 1e-3$ are used where applicable.*

however. Analysis of predicted masks from various models, some of which are shown in figure 11, shows that the base U-Net model tends to under-predict cilia, explaining the relatively high precision. Previous endeavors in cilia segmentation also revealed this pattern.

Conclusions

This paper highlights the current shortcomings of automated, deep-learning based segmentation models for cilia, specifically on the data provided to the Quinn Research Group, and provides two additional methods, Zero-Phase PCA Sphering (ZCA) and Sparse Autoencoders (SAE), for performing feature extracting augmentations with the purpose of aiding a U-Net model in segmentation. An analysis of U-Nets with various combinations of these feature extraction and parameters help determine the feasibility for low-level feature extraction in improving cilia segmentation, and results from initial experiments show up to 10% increases in relevant metrics.

While these improvements, in general, have been marginal, these results show that pre-segmentation based feature extraction methods, particularly the avenues explored, provide a worthwhile path of exploration and research for improving cilia segmentation. Implications internal to other projects within the research group sponsoring this research are clear. As discussed earlier, later pipelines of ciliary representation and modeling are currently being bottlenecked by the poor segmentation masks produced by base U-Nets, and the under-segmented predictions provided by the original model limits the scope of what these later stages may achieve. Better predictions hence tend to transfer to better downstream results.

These results also have significant implications outside of the specific task of cilia segmentation and modeling. The inherent problem that motivated an introduction of feature extraction into the segmentation process was the poor quality of the given dataset. From occlusion to poor lighting to blurred images, these are problems that typically plague segmentation models in the real world, where data sets are not of ideal quality. For many modern computer vision tasks, segmentation is a necessary technique to begin analysis of certain objects in an image, including any forms of objects from people to vehicles to landscapes. Many images for these tasks are likely to come from low-resolution imagery, whether that be satellite data or security cameras, and are likely to face similar problems as the given cilia dataset in terms of image quality. Even if this is not the case, manual labelling, like that of this dataset and convenient in many other instances, is prone to error and is likely to bottleneck results. As experiments have shown, feature extraction through SAE and ZCA maps are a potential avenue for improvement of such models and would be an interesting topic to explore on other problematic datsets.

Especially compelling, aside from the raw numeric results, is how soon composite pipelines began to produce usable masks on training data. As discussed earlier, most original U-Net models would take at least 40-50 epochs before showing any accurate predictions on training data. However, when feeding in composite SAE and ZCA data along with the original image, unusually accurate masks were produced within just a couple minutes, with usable results at 10 epochs. This has potential implications in scenarios such as one-shot and/or unsupervised learning, where models cannot train over a large datset.

Future Research

While this work establishes a primary direction and a novel perspective for segmenting cilia, there are many interesting and valuable directions for future planned research. In particular, a novel and still-developing alternative to the convolution layer known as a Sharpened Cosine Similarity (SCS) layer has begun to attract some attention. While regular CNNs are proficient at filtering, developing invariance to certain forms of noise and perturbation, they are notoriously poor at serving as a spatial indicator for features. Convolution activations can be high due to changes in luminosity and do not necessarily imply the *distribution* of the underlying luminosity, therefore losing precise spatial information. By design, SCS avoids these faults by considering the mathematical case of a "normalized" convolution, wherein neither the magnitude of the input, nor of the kernel, affect the final output. Instead, SCS activations are dictated purely by the *relative* magnitudes of weights in the kernel, which is to say by the *spatial distribution* of features in the input [Pis22]. Domain knowledge suggests that cilia, while able to vary greatly, all share relatively unique spatial distributions when compared to non-cilia such as cells, out-of-phase structures, microscopy artifacts, etc. Therefore, SCS may provide a strong augmentation to the backbone U-Net model by acting as an additional layer *in tandem with* the

already existing convolution layers. This way, the model is a true generalization of the canonical U-Net and is less likely to suffer poor performance due to the introduction of SCS.

Another avenue of exploration would be a more robust ablation study on some of the hyperparameters of the feature extractors used. While most of the hyperparameters were chosen based on either canonical choices [XXL+16] or through empirical study (e.g. ε for ZCA whitening), a more comprehensive hyperparameter search would be worth consideration. This would be especially valuable for the composite model since the choice of most optimal hyperparameters is dependent on the downstream tasks and therefore may be different for the composite model than what was found for the individual models.

More robust data augmentation could additionally improve results. Image cropping and basic augmentation methods alone provided minor improvements of just the base U-Net from the state of the art. Regarding the cropping method, an upper threshold for the percent of cilia per image may be worth implementing, as cropped images containing over approximately 90% cilia produced poor results, likely due to a lack of surrounding context. Additionally, rotations and lighting/contrast adjustments could further augment the data set during the training process.

Re-segmenting the cilia images by hand, a planned endeavor, will likely provide more accurate masks for the training process. This is an especially difficult task for the cilia dataset, as the poor lighting and focus even causes medical professionals to disagree on the exact location of cilia in certain instances. However, the research group associated with this paper is currently in the process of setting up a web interface for such professionals to "vote" on segmentation masks. Additionally, it is likely worth experimenting with various thresholds for converting U-Net outputs into masks, and potentially some form of region growing to dynamically aid the process.

Finally, it is possible to train the SAE and U-Net jointly as an end-to-end system. Current experimentation has foregone this path due to the additional computational and memory complexity and has instead opted for separate training to at least justify this direction of exploration. Training in an end-to-end fashion could lead to a more optimal result and potentially even an interesting latent representation of ciliary features in the image. It is worth noting that larger end-to-end systems like this tend to be more difficult to train and balance, and such architectures can fall into degenerate solutions more readily.

REFERENCES

[DvBB+21] Cenna Doornbos, Ronald van Beek, Ernie MHF Bongers, Dorien Lugtenberg, Peter Klaren, Lisenka ELM Vissers, Ronald Roepman, Machteld M Oud, et al. Cell-based assay for ciliopathy patients to improve accurate diagnosis using alpaca. *European Journal of Human Genetics*, 29(11):1677 |–| 1689, 2021. doi:10.1038/s41431-021-00907-9.

[Ish17] Takashi Ishikawa. Axoneme structure from motile cilia. *Cold Spring Harbor perspectives in biology*, 9(1):a028076, 2017. doi:10.1101/cshperspect.a028076.

[LjWD19] Hui Li, Xiao jun Wu, and Tariq S. Durrani. Infrared and visible image fusion with resnet and zero-phase component analysis. *Infrared Physics & Technology*, 102:103039, 2019. doi:https://doi.org/10.1016/j.infrared.2019.103039.

[LMZ+18] Charles Lu, M. Marx, M. Zahid, C. W. Lo, Chakra Chennubhotla, and Shannon P. Quinn. Stacked neural networks for end-to-end ciliary motion analysis. *CoRR*, 2018. doi:10.48550/arXiv.1803.07534.

[LWL+18] Fangzhao Li, Changjian Wang, Xiaohui Liu, Yuxing Peng, and Shiyao Jin. A composite model of wound segmentation based on traditional methods and deep neural networks. *Computational intelligence and neuroscience*, 2018, 2018. doi:10.1155/2018/4149103.

[Pis22] Raphael Pisonir. Sharpened cosine distance as an alternative for convolutions, Jan 2022. URL: https://www.rpisoni.dev.

[QZD+15] Shannon P Quinn, Maliha J Zahid, John R Durkin, Richard J Francis, Cecilia W Lo, and S Chakra Chennubhotla. Automated identification of abnormal respiratory ciliary motion in nasal biopsies. *Science translational medicine*, 7(299):299ra124 |–| 299ra124, 2015. doi:10.1126/scitranslmed.aaa1233.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, 2015. doi:10.48550/arXiv.1505.04597.

[WMBL19] Nicholas Watters, Loïc Matthey, Christopher P. Burgess, and Alexander Lerchner. Spatial broadcast decoder: A simple architecture for learning disentangled representations in vaes. *CoRR*, 2019. doi:10.48550/arXiv.1901.07017.

[XXL+16] Jun Xu, Lei Xiang, Qingshan Liu, Hannah Gilmore, Jianzhong Wu, Jinghai Tang, and Anant Madabhushi. Stacked sparse autoencoder (ssae) for nuclei detection on breast cancer histopathology images. *IEEE Transactions on Medical Imaging*, 35(1):119–130, 2016. doi:10.1109/TMI.2015.2458702.

[ZRS+20] Meekail Zain, Sonia Rao, Nathan Safir, Quinn Wyner, Isabella Humphrey, Alex Eldridge, Chenxiao Li, BahaaEddin AlAila, and Shannon Quinn. Towards an unsupervised spatiotemporal representation of cilia video using a modular generative pipeline. In *Proceedings of the Python in Science Conference*, 2020. doi:10.25080/majora-342d178e-017.