Proceedings of the 18th Python in Science Conference

July 8 - July 14 • Austin, Texas

PROCEEDINGS OF THE 18TH PYTHON IN SCIENCE CONFERENCE

Edited by Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe.

SciPy 2019
Austin, Texas
July 8 - July 14, 2019

ISSN:2575-9752
https://doi.org/10.25080/Majora-7ddc1dd1-026

ORGANIZATION

Conference Chairs

SERGE REY, Arizona State University
CORRAN WEBSTER, Enthought, Inc.

Program Chairs

GIL FORSYTH, George Washington University
NELLE VAROQUAUX, Berkeley Institute for Data Science

Communications

PAUL IVANOV, Bloomberg

Birds of a Feather

MATTHIAS BUSSONNIER, University of California, Merced
LINDSEY HEAGY, University of California, Berkeley

Proceedings

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

Financial Aid

CELIA CINTAS, IBM Research Africa
SCOTT COLLIS, Argonne National Laboratory
ERIC MA, Novartis Institutes for Biomedical Research

Tutorials

ALEXANDRE CHABOT-LECLERC, Enthought, Inc.
ALLEN DOWNEY, Franklin W. Olin College of Engineering
MIKE HEARNE, USGS

Sprints

RYAN MAY, University Corporation for Atmospheric Research
KUYA TAKAMI, Enthought, Inc.

Diversity

NATHAN GOLDBAUM, University of Illinois
MELISSA MENDONCA, Federal University of Santa Catarina

Activities

KYLE NIEMEYER, Oregon State University
JULIA PASQUARELLA, Enthought

Sponsors

JILL COWAN, Enthought

Financial

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

Logistics

JILL COWAN, Enthought, Inc.

Proceedings Reviewers

Alberto Antonietti
Alejandro Weinstein
Amir Kalighi
Angelos Krypotos
Ankur Ankan
Awalin Sopan
Brian McFee
Catherine Ordun
Christine Choirat
Cyrus Harrison
David Nicholson
Filipe Fernandes
Haruna Abdu
Jaime Arias Almeida
James Bednar
Kirtan Dave
Laura Kahn
Mahdi Sadjadi
Marc Wouts
Mark Fenner
Matt Craig
Matthew Seal
Meghann Agarwal
Nicholas Malaya
Patrick Huck
Scott Sievert
Senol Piskin
Shuai Tang
Stefan van der Walt
Tania Lorido Botran
Yingwei Yu
Zach Golkhou

Accepted Talk Slides

To a Billion and Beyond: How to Visually Explore, Compare and Share Large Quantitative Datasets with HiGlass, Peter Kerpedjiev, and Nezar Abdennur, and Fritz Lekschas
doi.org/10.25080/Majora-7ddc1dd1-01c
Inside NumPy: Preparing for the next decade, Ralf Gommers, and Sebastian Berg, and Matti Picus, and Tyler Reddy, and Stéfan van der Walt, and Charles Harris
doi.org/10.25080/Majora-7ddc1dd1-01d
Turning HPC Systems into Interactive Data Analysis Platforms using Jupyter and Dask, Anderson Banihirwe, and Matthew Rocklin, and Joseph Hamman, and Julia Kent, and Kevin Paul
doi.org/10.25080/Majora-7ddc1dd1-01e
Visualization of Bioinformatics Data with Dash Bio, Shammamah Hossain
doi.org/10.25080/Majora-7ddc1dd1-01f
Optimizing Python-Based Spectroscopic Data Processing on NERSC Supercomputers, Laurie A. Stephey, and Rollin C. Thomas, and Stephen J. Bailey
doi.org/10.25080/Majora-7ddc1dd1-020
Building and Replicating Models of Visual Search Behavior with Tensorflow and the Scientific Python Stack, David Nicholson
doi.org/10.25080/Majora-7ddc1dd1-021

Accepted Posters

PyMeasRF: Automating RF Device Measurements Using Python, Jackson Anderson, and Dana Weinstein
doi.org/10.25080/Majora-7ddc1dd1-014
A Pythonic Equivalent Circuit Model for Battery Research, Gavin Wiggins, and Srikanth Allu, and Hsin Wang
doi.org/10.25080/Majora-7ddc1dd1-015
An intelligent shopping list based on the application of partitioning and machine learning algorithms, Nadia Tahiri, and Bogdan Mazoure, and Vladimir Makarenkov
doi.org/10.25080/Majora-7ddc1dd1-016
Python Workflow for High-Fidelity Modeling of Overland Hydrocarbon Flows with GeoClaw and Cloud Computing, Pi-Yueh Chuang, and Tracy Thorleifson, and Lorena A. Barba
doi.org/10.25080/Majora-7ddc1dd1-017
The PyDataWeaver: A data integration platform, Henry Senyondo, and Andrew Zhang, and Ethan P. White
doi.org/10.25080/Majora-7ddc1dd1-018
pyhf: a pure Python statistical fitting library for High Energy Physics with tensors and autograd, Matthew Feickert, and Lukas Heinrich, and Giordon Stark, and Kyle Cranmer
doi.org/10.25080/Majora-7ddc1dd1-019
Using Python to Model Biomass Pyrolysis Reactors, Gavin Wiggins
doi.org/10.25080/Majora-7ddc1dd1-01a
Class-based ODE solvers and event detection in SciPy, David R Hagen, and Nikolay Mayorov
doi.org/10.25080/Majora-7ddc1dd1-01b

SciPy Tools Plenaries

SciPy: not just a conference!, Matt Haberland
doi.org/10.25080/Majora-7ddc1dd1-022

Lightning Talks

The Mouse Aging Cell Atlas: cell biology meets Python, The Tabula Muris consortium, and Angela Oliveira Pisco, and Nicholas Schaum, and Aaron McGeever, and Jim Karkanias, and Norma F. Neff, and Spyros Darmanis, and Tony Wyss-Coray, and Stephen R. Quake
doi.org/10.25080/Majora-7ddc1dd1-023
Automated Annotation of Animal Vocalizations, David Nicholson
doi.org/10.25080/Majora-7ddc1dd1-024
US Research Software Sustainability Institute (URSSI) Pilot 'Summer' School, Kyle E. Niemeyer, and Jeffrey Carver, and Karthik Ram
doi.org/10.25080/Majora-7ddc1dd1-025

SCHOLARSHIP RECIPIENTS

MRIDUL SETH, UCLouvain, Belgium
IVAN OGASSAVARA, Quansight
FILIPE FERNANDES, NOAA/IOOS
SIMON CROSS, Prodigy Finance
JAMIE ARIAS, Institute Galilée - Université Paris 13
RYAN AVERY, University of California, Santa Barbara
TZU-CHI YEN, University of Colorado, Boulder
WANJUN ZHANG, Code Park Houston
ANDY GAWORECKI, Clear Creek ISD

JUMP TRADING AND NUMFOCUS DIVERSITY SCHOLARSHIP RECIPIENTS

MELISSA WEBER MENDONÇA, Federal University of Santa Catarina
GAJENDRA DESHPANDE, KLS Gogte Institute of Technology
PARUL SETHI, Centre for Neuroscience, Indian Institute of Science
SERAH NJAMBI, The Carpentries
NADIA TAHIRI, UQAM / Plotly
DEBORAH HANUS, Sparrow
VERONICA HANUS,
ABIGAIL SEARFOSS, Vanderbilt University
HANNAH AIZENMAN, Matplotlib
MARGARET NG, University of Illinois, Urbana Champaign
BWIHANGANE BIRINDWA, Evangelical University in Africa

CONTENTS

Accelerating the Advancement of Data Science Education

Eric Van Dusen‡*, Anthony Suen‡, Alan Liang‡, Amal Bhatnagar‡

Abstract—We outline a synthesis of strategies created in collaboration with 35+ colleges and universities on how to advance undergraduate data science education on a national scale. The four core pillars of this strategy include the integration of data science education across all domains, establishing adoptable and scalable cyberinfrastructure, applying data science to non-traditional domains, and incorporating ethical content into data science curricula. The paper analyzes UC Berkeley's method of accelerating the national advancement of data science education in undergraduate institutions and examines the recent innovations in autograders for assignments which helps scale such programs. The conversation of ethical practices with data science are key to mitigate social issues arising from computing, such as incorporating anti-bias algorithms. Following these steps will form the basis of a scalable data science education system that prepares undergraduate students with analytical skills for a data-centric world.

Index Terms—data science education, autograding, undergraduate institutions

Introduction

Data science is a burgeoning field that is quickly being adopted across all domains and sectors. Undergraduate data science education initiatives have been growing rapidly, but also largely in an uncoordinated manner. Programs are often developed and implemented in silos, leading to duplication of efforts and differences in pedagogical approaches and course quality. Furthermore, while a number of curriculum guidelines for degrees in data science have been proposed, opportunities for engaging in pedagogical exchanges and sharing resources remain rare. Without a common knowledge base of resources and platform for undergraduate support, many institutions have encountered pedagogical and infrastructural barriers in setting up Python centric data science curricula across campus.

Stakeholders around the country previously identified the necessity of gathering data science enthusiasts and discussing its implementation in institutions. The importance of defining data science curriculum guidelines has been the subject of numerous workshops and meetings such as the "Workshop on Theoretical Foundations of Data Science," "The Park City Math Institute 2016 Summer Undergraduate Faculty Program," and "Envisioning the Data Science Discipline: The Undergraduate Perspective." While these workshops produced comprehensive guidelines on the structure of the programs, the actual content and teaching modalities remain unclear.

On June 24-27, 2019, the Division of Data Sciences at the UC Berkeley hosted the 2nd National Workshop on Data Science Education which brought together nearly 70 faculty from a diverse range of higher-education institutions on at different stages of data science education. As a pioneer in undergraduate data science education, UC Berkeley shared its comprehensive set of open-license and open-source resources that range from teaching materials to cloud infrastructure at the workshop.

The workshop sought to build a national community of practice around undergraduate Data Science education by focusing around four primary areas:

1) Examining the *Foundations of Data Science (Data 8)* course - How does the content to pedagogical methods of Data 8 integrates various disciplines through the use of computational and inferential thinking.

2) Showcase the infrastructural platform that Berkeley has developed for its courses, and empower participants to use its many open-source components to overcome the *financial and technical barriers.*

3) Applying Modular Data Science education content into various disciplinary fields past the scope of that from a traditional computer science or statistics courses provide.

4) Implementing Ethical Content into *data science courses and examining how such integration could work.*

Establishing Foundational Course that Serves the Entire Campus

Data science can touch all different genres and disciplines of academia. The proliferation of affordable computational capacity, migration of publishing channels to the internet, advanced sensing technology, and other data collection methods has led to the possibility of data science in almost every area of scientific endeavor. Applications of data science have created opportunities to teach students programming, statistical techniques, and other computational methodologies earlier in their academic careers to expand the academic possibilities within their chosen area of focus. Current examples of data science being integrated into other disciplines include randomized controlled trials in Development Economics published through open data repositories and the integration between law and data technologies.

An essential step toward creating a successful campus wide Data Science program is creating a campus wide introductory-level data science course that utilizes the Python programming

* *Corresponding author: ericvd@berkeley.edu*
‡ *University of California, Berkeley*

language available to students of all academic disciplines. Using existing introductory computer science and statistics courses in place of a foundational data science course slows students' learning and limits the audience. Such a course also allows students to explore the realm of data science at an introductory level, so they can understand the basic concepts using custom made Data Science "Tables" library without getting lost in the more complex syntax of Pandas and transition into statistics and computer science, as many students do not have prior coding experience. In 2015, UC Berkeley launched Data 8 *Foundations of Data Science* for its undergraduate students. With the course centered upon students learning inferential thinking, computational thinking, and real-world relevance, students learn how to apply such statistical or computer programming techniques onto non-traditional fields through economic and geographical data and social issues. Using the Python datascience package, students work with real-world datasets to ask questions and find answers.

Through Data 8, core foundational facilitators now know to find connections to other departments and stakeholders. In fielding an entry-level data science course and fitting it into the curriculum, it's crucial to engage with faculty across a variety of disciplines in an inclusive, supportive way and a spirit of partnership. Berkeley has launched and taught many different connector courses that use Data 8 as a prerequisite. The Data Science students having classes from a non-traditional field allows students to find ways to apply their learning from Data 8 onto other domains.

Setting Campus Wide Educational Cyber-Infrastructure

Implementation of a data science course like Data 8 across the entire campus requires universities and institutions to develop capacity in on-demand cyber-infrastructure to support their educational goals. Local computation is not ideal, as it is harder to scale when the number of courses and students increases. For many institutions, the ability to set up the necessary support systems for JupyterHub or other infrastructure is beyond the expertise of a single course instructor, who already has to distribute their finite time in planning lesson outlines and curriculum. Institutional IT staff members would have to obtain additional training, which would vary across institutions to better fit the differing needs and implementations of the data science courses and can be too costly. For many small institutions and universities, this proves to be a major barrier in course delivery. The development of regional or national cloud-based computing solutions that can serve individual educational institutions is needed.

Universities must invest resources into developing data science educational infrastructure like JupyterHub, a platform not many universities have, that differs from research cyber-infrastructure. The two have different goals, resource needs, deployment timelines, cost and pricing of models, and broad access mandates. Data science educational infrastructure is deployed for relatively low resource use by a large number of relatively unsophisticated users. Making the data science infrastructure accessible requires establishing three components. At UC Berkeley, the core components include setting up a campus wide JupyterHub, integration with existing campus Learning Management Systems (LMS), e.g. Canvas (https://www.instructure.com/), and utilizing autograder technology.

Autograding technology is essential to the scalability of data science education and alleviates substantial work for large classes at UC Berkeley, such as *Data 8: Foundations of Data Science* and *Data 8X*, its massive open online course, or MOOC, version, which sees more than 1,500 students per semester and 75,000 students enrolled respectively. Currently, UC Berkeley uses various grading systems even within its own data science courses. *Data 8* utilizes ok.py, a Berkeley developed solution that has a plethora of features for large and diverse computer science and data science classes. However, this comes with a complexity cost for instructors who only need a subset of these features and sysadmins operating an okpy server installation [Suen18]. On the other hand, Data 100, the upper division core data science course, utilizes *nbgrader*, an open source grading solution built for Jupyter Notebooks. On Data 8X, the newly developed *gofer grader* is used to solely address the needs of a MOOC course and retains similar aspects from Data 8's grading system. The *gofer grader* is relatively new and has run into issues relatively frequently. Yet, it asynchronously supports hundreds of students' grading concurrently

To mitigate high individual institutional infrastructure startup costs, a national educational cyber-infrastructure strategy with industry and universities collaboration is required. Options include leveraging the existing four regional Big Data Innovation Hubs, which can provide access to cloud resources, partners and expertise or increase utilization of currently free industry platforms like Google Colab and Azure Notebooks. To maximize learning within any pilot program, local staff at a given institution would need to be trained and partake in the beta testing of such a system to document problems and best practices. Successful implementation of data science courses across certain locations might lead to partnerships across and within institutions, allowing for successful techniques to be communicated across all partners and similar curriculum modeling to exist for consistency.

All of this infrastructure is crucial for creating, deploying, and grading data science homework and lab assignments. Having this educational cyber-infrastructure is more efficient than local infrastructure, as instructors can teach students for many, the system holds all the necessary material, simplifies data management and analysis, and visualizes data for instructors. Before Berkeley launched its integrated system, the teaching faculty found it difficult to efficiently scale courses at the rate of their increasing interest. Berkeley's adoption of JupyterHub has allowed more than 1,600 students to enroll in Data 8 for its Spring 2019 iteration, a historic milestone that would not have been possible absent Berkeley's educational cyber-infrastructure.

Creating and Incorporating Modular Data Science Content

There are two main concerns when modularizing data science content: *Having just one introductory data science class is not enough to warrant an entire data science curricula, and creating a sustainable model that supports the data science curricula is challenging for newly adopting institutions.*

Implementing and integrating the new course to fit in the overall academic curriculum is critical for seamless student experience in data science. UC Berkeley's Division of Data Sciences has also supported the creation of data science content for inserting in other types of (usually non-data science) courses in self-contained "Modules" that can showcase aspects of data science to a different audience. Some examples of modules that students can take include Linguistics 110: *Introduction to Phonetics and Phonology,* Sociology 130 AC: *Neighborhood Mapping,* and Econ 101B: *Macroeconomics*. Developing and implementing such modules allow students to experience data-driven techniques and scientific computing through Python.

Because data science serves functions in a vast array of interdisciplinary fields of study, the ability to modify the introductory course and tailor it to fit in with the current institutional curriculum will go a long way in communicating the relevance of the field to students taking the course. This process will need time for planning and preparation before the actual steps for integration can start. In addition, faculty across different departments should collaborate to explore the possibility of connector courses or incorporation of data science in each others' subjects. Connector courses are supplemental courses which build on the introductory data science course by using similar statistical and computational techniques across different disciplines, such as business, biology, and geography. Berkeley has offered 27 different connector courses since their launch in 2015. To alleviate the burden of redistributing finances and to increase funding, faculty might have to reallocate their time to develop and adopt a new curriculum. To mitigate increasing startup costs, Berkeley has hired graduate students and even undergraduate students who previously excelled at that class to assist in teaching efforts. Incorporating on-campus talent, such as previous students, creates a robust data science culture on campus that is easy to spread among the student population.

To successfully adopt a data science modules curricula, we propose creating a platform to share teaching resources that is available to anyone in the community. Such a platform could be modeled on the popular Data8 public organization (https://github.com/data-8) and the site hosting Data Carpentry lessons (https://datacarpentry.org/lessons/). The principal functions of this platform are to share teaching resources such as use cases (datasets and accompanying analyses), open source textbooks or modules, and programs used to facilitate data science education. National Workshop on Data Science Education proves that the design of the courses and the planning of the material and activities is key. Berkeley's Data 8's success in reaching up to 1,500 students within its first few iterations attests to the importance of curriculum innovation and pedagogical methods. Having staff with technical skills to support the computer infrastructure and collaborative support with nearby/sister institutions who can share best practices and resources makes this model even more successful. Developing collaborative, modularized open-source teaching materials, such as the books used in Data 8 and Data 100, allows other institutions to more easily implement curricula for themselves.

Recently, Berkeley has been sharing such resources with institutions interested in adopting a data science curriculum. By sharing access to textbooks, lecture and lab materials, and similar resources, about 15 domestic and 10 international institutions have adopted Data 8 or a similar course or program. Most questions potential partnering institutions had regarded logistics, course topics, and infrastructure, which were resolved once given access to shared resources. Such partnering institutions range from community colleges to Ivy League universities indicating the widespread approval of Data 8's goals, implementation, and adaptability. Berkeley's cross-campus collaboration proves that transparency and communication is key to start and scale undergraduate data science programs across the world and increase Python literacy.

Integrating The Teaching of Ethics Into Data Science Courses

As data come to structure more and more aspects of our lives, the potential impact of data science on individuals and societies looms ever larger. For this reason, it is critical that data scientists understand the social worlds from which their data are drawn and in which their science intervenes. They must be trained to recognize the ethical implications of their work and act accordingly. The ethics of data science are social, individual, and contextual rather than linear. Ethical content can be incorporated into data science curricula both by integrating ethical topics into existing data science courses and by including ethically-focused courses to data science degree programs. The first approach may be better suited to the ethical questions that individual data scientists encounter in their daily work, while the second may be better suited to the broader issues raised by the growing role of data and algorithms in society as a whole. For example, ethical questions arise at every step of the data science life cycle. Where data science courses teach professional competencies of statistics, computer science, and various content areas, they can also introduce students to the ethical standards of research and practice in those domains [NASEMS18]. Some data science textbooks already address such issues as misleading data visualizations, p-hacking, web scraping, and data privacy [Baumer17].

A recent trend in incorporating such ethical practices includes incorporating anti-bias algorithms in the workplace. Starting from the beginning of their undergraduate education, UC Berkeley students can take *History 184D: Introduction to Science, Technology, and Society: Human Contexts and Ethics of Data,* which covers the implications of computing, such as algorithmic bias. Additionally, students can take *Computer Science 294: Fairness in Machine Learning,* which spends a semester in resisting racial, political, and physical discrimination. Faculty have also come together to create the Algorithmic Fairness and Opacity Working Group at Berkeley's School of Information that brainstorms methods to improve algorithms' fairness, interpretability, and accountability. Implementing such courses and interdisciplinary groups is key to start the conversation within academic institutions, so students can mitigate such algorithmic bias when they work in industry or academia post-graduation.

Databases and algorithms are socio-technical objects; they emerge and evolve in tandem with the societies in which they operate [Latour90]. Understanding data science in this way and recognizing its social implications requires a different kind of critical thinking that is taught in data science courses. Issues such as computational agency [Tufekci15], the politics of data classification and statistical inference [Bowker08], [Desrosieres11], and the perpetuation of social injustice through algorithmic decision making [Eubanks19], [Noble18], [ONeil18] are well known to scholars in the interdisciplinary field of science and technology studies (STS), who should be invited to participate in the development of data science curricula. STS or other courses in the social sciences and humanities dealing specifically with topics related to data science may be included in data science programs.

Including training in ethical considerations at all levels of society and all steps of the data science workflow in undergraduate data science curricula could play an important role in stimulating change in industry as our students enter the workforce, perhaps encouraging companies to add ethical standards to their mission statements or to hire chief ethics officers to oversee not only day-to-day operations but also the larger social consequences of their work.

Summary & Vision

We envision a world where all students can learn ethical data-driven techniques regardless of their domain and can manipulate

data to find better solutions to problems. To do that requires a four part strategy involving creating a campus wide foundational data science course, the modularization of data science course content to integrate it with courses in existing domains, the scalable cloud infrastructure power it all, and the human context and ethics content to reign in misuse of data & artificial intelligence. Integrating Python across different fields exposes students to learning programming in areas they would not have previously expected. These strategies will accelerate the creation of a space for Data Science to exist as a cross-campus endeavor and engage faculty and students in different departments

REFERENCES

[Baumer17] Baumer, B. S., Kaplan, D. T., & Horton, N. J. (2017). Modern Data Science with R. Retrieved from http://mdsr-book.github.io/

[Bowker08] Bowker, G. C., & Star, S. L. (2008). Sorting things out: Classification and its consequences. Cambridge, MA: MIT Press.

[Desrosieres11] Desrosieres, A. (2011). The politics of large numbers: A history of statistical reasoning. Cambridge, MA: Harvard University Press.

[Eubanks19] Eubanks, V. (2019). AUTOMATING INEQUALITY: How high-tech tools profile, police, and punish the poor. PICADOR.

[Latour90] Latour, B. (1990). Technology is society made durable. The Sociological Review, 38(1), supplement, 103-131. Retrieved from http://www.bruno-latour.fr/sites/default/files/46-TECHNOLOGY-DURABLE-GBpdf.pdf

[NASEMS18] National Academies of Sciences, Engineering, and Medicinemies of Sciences. (2018, May 02). Data Science for Undergraduates: Opportunities and Options. Retrieved from https://doi.org/10.17226/25104

[Noble18] Noble, S. U. (2018). Algorithms of oppression how search engines reinforce racism. New York: New York University Press.

[ONeil18] ONeil, C. (2018). Weapons of math destruction: How big data increases inequality and threatens democracy. London: Penguin Books.

[Suen18] Suen, A., Norén, L., Liang, A., & Tu, A. (2018). Equity, Scalability, and Sustainability of Data Science Infrastructure. Proceedings of the 17th Python in Science Conference. doi:10.25080/majora-4af1f417-002

[Tufekci15] Tufekci, Z. (2015). Algorithmic harms beyond Facebook and Google: Emergent challenges of computational agency. Colorado Technology Law Journal, 203-218. Retrieved from https://ctlj.colorado.edu/wp-content/uploads/2015/08/Tufekci-final.pdf.

Case study: Real-world machine learning application for hardware failure detection

Hongsup Shin‡*

Abstract—When designing microprocessors, engineers must verify whether the proposed design, defined in hardware description language, does what is intended. During this verification process, engineers run simulation tests and can fix bugs if the tests have failed. Due to the complexity of the design, the baseline approach is to provide random stimuli to verify random parts of the design. However, this method is time-consuming and redundant especially when the design becomes mature and thus failure rate is low. To increase efficiency and detect failures faster, it is possible to train machine learning models by using previously run tests, and assess the likelihood of failure of new test candidates before running them. This way, instead of running random tests agnostically, engineers use the model prediction on a new set of test candidates and run a subset of them (i.e., "filtering" the tests) that are more likely to fail. Due to the severe imbalance (1% failure rate), I trained an ensemble of supervised (classification) and unsupervised (outlier detection) models and used the union of the prediction from both models to catch more failures. The tool has been deployed in an internal high performance computing (HPC) cluster early this year, as a complementary workflow which does not interfere with the existing workflow. After the deployment, I found performance instability in post-deployment performance and ran various experiments to address the issue, such as by identifying the effect of the randomness in the test generation process. In addition to introducing the relatively new data-driven approach in hardware design verification, this study also discusses the details of post-deployment evaluation such as retraining, and working around real-world constraints, which are sometimes not discussed in machine learning and data science research.

Index Terms—hardware verification, machine learning, outlier detection, deployment, retraining, model evaluation

Introduction

Simulation-based hardware verification

Hardware verification is the process of checking that a given design correctly implements the specifications, which are the technical description of the computer's components and capabilities. It is recognized as the largest task in silicon development and as such has the biggest impact on the key business drivers of quality, schedule and cost. In the computer hardware design cycle, microprocessor manufacturing companies often spend 60-70% of the cycle dedicated to the verification procedure. Traditionally, two techniques have been used: formal and simulation-based (random-constraint) methods [Ioa12]. The former adopts a mathematical approach such as theorem proving and requirement checks [Wil05], which provides exhaustiveness but doesn't scale well with design complexity. Due to the exponentially-growing design complexity, the more widely used approach is the simulation-based testing, which simulates a design by providing stimuli to tests. These stimuli can be considered as arbitrary values that control certain functionalities of the design that were expressed in hardware description language such as whether to turn on or off a specific setting. During simulation-based testing, verification engineers provide a set of constraints to stimuli so that they can direct tests toward a certain direction. However, it is not easy to target certain design parts deterministically and engineers often depend on previous knowledge or intuition.

Failures (bugs) in hardware verification

Hardware verification can be compared to unit testing in software engineering, especially since design functionalities are realized in hardware description language (HDL) like Verilog. Similar to software testing, hardware verification process involves checking whether simulations of the code written in HDL with a set of given input values (i.e., tests with certain inputs), show desirable behavior. If a test returns undesirable output, it is considered as a failure (bug). To fix the failures, engineers modify the HDL source code such as by fixing "assign" statements or by correcting or adding conditions (e.g., "if" statements), and so on [Sud08]. The HDL-level hardware verification is one of the many steps in hardware testing, which precedes physical design implementation. This low-level verification is a critical step in hardware testing because fixing a bug in a higher level (e.g., in physical design or even in a product) is more costly and challenging because it is hard to identify which previous steps have bugs.

Previous machine-learning based approach

The ultimate goal of hardware verification is to have a (close-to) failure-free design. From the simulation-based testing perspective, this is an exploration problem where machine learning can be useful. For instance, reinforcement learning algorithms can be used to explore the complex space of test stimuli by learning a reward function [Ioa12]. However, this approach is not feasible because the simulation-based testing is non-deterministic and intractable, which makes it difficult to estimate the level of stochasticity. This is mainly because the motivation for the simulation-based approach is randomization, often implemented in multiple steps (i.e., a value in an input setting randomizes a value in the next step, which then randomizes a value of a different setting in the following step, etc.). The testing tools have not been built to track these setting values and the information on

* Corresponding author: hongsup.shin@arm.com
‡ Arm Research

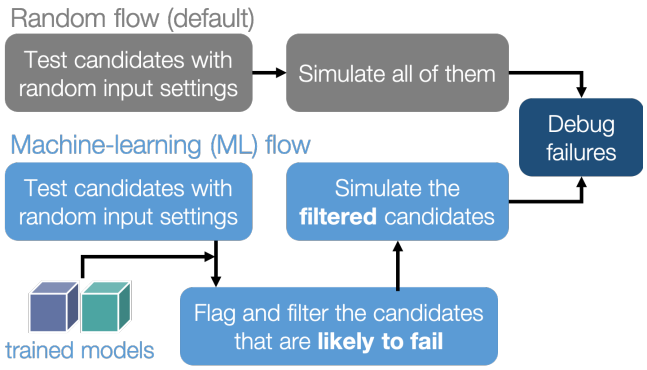


Fig. 1: *Overview of the prototype pipeline. Top: the existing workflow (the randomized testing). Bottom: the complementary machine learning (ML) flow. By default, engineers run all tests that are randomly generated. In the ML flow, before running tests, the test candidates (input settings) are shown to the models first. The models then flag which tests are likely to fail. In the end, engineers can run the flagged test candidates only. In the final deployed version, approximately 1000 test candidates are provided to the ML flow, which passed about 400 tests. This corresponds to 10% of the number of tests in the top flow. The cubes correspond to the pre-trained machine learning models (blue: a supervised model, green: an unsupervised model).*

probability distributions used in the randomization process have been left out. To address this, a few studies [Bar08], [Fin09] adopted a probabilistic approach but they failed to mention actual implementation in production cycle and scalability issue. The majority of the previous research on hardware verification with the simulation-based testing approach has focused on supervised learning [Mam16], [Bar08], [Wag07] and evolutionary algorithms [Ber13], [Cru13]. [Mam16] has shown a study that is the closest to this study in nature but the authors focused on high-level instruction set simulator (ISS), which generates instructions at a higher level (related to hardware performance, a high-level metric) than the design level.

Simulation-based testing in practice

In practice, engineers build a testbench to house all the components that are needed for the verification process: test generator, interface, driver, monitor, model, and scoreboard. To run tests, verification engineers define a set of values as *input settings*, which can be compared to input arguments to a function. These values are passed to the test generator, and under certain constraints, a series of subsequent values that stimulate various parts of the design are *randomly generated*. This information is then passed to the interface through the driver. The interface interacts with a design part (register-transfer level (RTL) design written in HDL) and then the returned output is fed into the monitor. To evaluate the result, the desirable output should be retrieved. This information is stored in the model, which is connected to the driver. A test is identified as a failure when the desirable output from the driver (through the model) and the output from the monitor do not match. In addition to the binary label of pass or failure, the testbench also returns a log file of the failure, if the test has failed. This log contains detailed information of the failure. Each failure log is encoded as an 8-digit hexadecimal code by a hash function. This code is called *unique failure signature (UFS)*. In general, instead of inspecting every failure log, engineers are more interested in maximizing the number of UFS that are collected after a batch of tests. Collecting a large number of UFS means failures with a great variety have been hunted down. Having a larger variety of failures is important because it means the tests have explored various parts of the design and thus, it's likely to discover failures associated with rare edge cases or problems overlooked before. Once a new UFS is found, engineers starts a debugging process to fix the failure.

Random generation of the test settings in the test generator is used to run a batch of tests automatically almost daily to explore random parts of the design with efficiency. In practice, engineers run tests with certain input settings and collect the results after the tests are simulated. The way that engineers control the input settings varies widely. In an extreme case, they only control the seed number of a pseudo-random number generator in the test generator for the entire set of the input settings of test candidates. Normally for a test, engineers have a set of input settings, not just the seed, which either turns a setting on and off or controls stochastic behavior a setting by defining what kind of values the setting can take. For instance, if a certain input setting has a string value of *"1-5"*, it indicates that the actual stimulus that goes into the simulation can be *any integer from 1 to 5*. Unfortunately, the testbench does not track this information and it is not possible to know which value ended up getting chosen eventually. Hence, it is extremely challenging to guide a testbench to generate a specific value of the input settings. *This is why building a machine learning model is challenging because two tests with the exact same values of an input setting can result in two different outcomes.* Additionally, engineers make changes to the design almost every day, which includes a new implementation or modification in the design, or bug fixes. This affects the test behavior and in turn, data generation process, which implies that the data distribution can potentially change almost daily (i.e., frequent data drift).

Working around the stochastic test generation

This situation requires a unique approach. It is impossible to eliminate randomness in the test generation step, which makes it difficult to guide testbench to test specific input values or parts of the system (cf. it is possible to target a specific module but the process is still not deterministic). Instead, we leave the inputs to be generated randomly and filter them afterward. By using the labeled data from previous tests (i.e., tests that were already simulated), a machine learning model (classifier) can be trained to predict whether a test will fail or pass with a given set of input settings. Then, it is possible to provide a large set of test *candidates* (a number of tests with random input setting values, i.e., providing the new input values) to the trained model that assesses which subset of the test candidates will fail. This way, it is possible to run the subset of tests only, instead of running the entire test candidates agnostically. This can bring cluster savings and make the verification process more efficient. However, the existing simulation-based testing with random constraints *should remain* because we still have to explore new design parts, which in turn provides new training data for model update. Hence, two parallel pathways can be proposed (Fig. 1); one with the default randomized testing and the other with machine learning models, where an additional set of test candidates are provided and then only the tests flagged by the models are filtered and run. This way, it is possible to continue collecting novel data from the first pathway to explore a new input space while utilizing the data from previous tests via the ML flow.

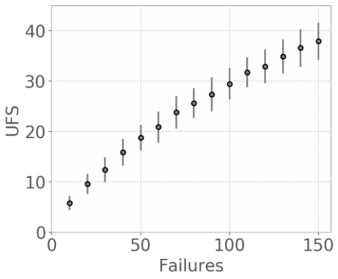


Fig. 2: *Relationship between the number of failures (x axis) and the number of unique fail signatures (UFS) on the y axis (mean and standard error). To generate the error bar, I ran 100 simulations where in each simulation, I draw N failed tests among a pool of 250k tests and counted the number of UFs. The more failures occur, the more UFS are found.*

Post-deployment analysis

I used both supervised and unsupervised models to address the severe class imbalance problem and used the union of the prediction from both models. This means, a test is predicted to fail when at least one of the two models predict it will fail. With this approach, for a set of independent testing datasets, it was possible to find 80% of unique failure signatures (Fig. 3) by running only 40% of tests on average, compared to running tests based on the original simulation-based method. The tool has been deployed in production since early this year in our internal cluster as a part of daily verification workflow, which is used by verification engineers in the production team. It is not common in both machine learning and hardware verification literature to find how suggested models perform in a real-world setting. Often machine learning studies show performance based on a single limited dataset or commonly used benchmark datasets. In this paper, I address this and attempt to provide practical insights to the post-deployment process such as decisions regarding the automation of model retraining and addressing randomness in the post-deployment period.

Methods

Data

Simulation-based testing is run almost every day via a testbench. Every simulated test and outcome (i.e., test success (pass or failure) and unique failure signature (UFS) if a test has failed) are stored in a database. To address the issue of data drift over time, two datasets are collected. The first dataset ("snapshot") is generated from a same version of testbench (115k tests). Model evaluation with this dataset provides information on the baseline model performance when data doesn't change over time. For the second set, a month's worth of data (ca. 6k tests per day) is collected. The second dataset ("1-month") is used specifically to simulate retraining scenarios and to challenge our model for every-day changes in the testbench (150k). Both datasets are from a specific unit of a microprocessor with a specific test scenario. The input dataset has individual tests as rows and test settings (stimuli) as columns. These settings are specified by verification engineers. The total number of settings are in the range of several hundreds. The output dataset has tests as rows and two columns, one for the pass-failure binary label and the other for the unique failure signatures of the failed tests.

Data preprocessing

The input data was preprocessed based on the domain knowledge of the verification engineers. In the raw data, roughly 70% of the data was missing, which corresponds to input settings that were not modified from the defaults. Using a software analogy, this is similar to not having to specify an input argument value in a function, if it already has a default value for that argument. The engineers were able to obtain the default values, which fixed the missing data issue. There were about 20% object (i.e., non-numerical) columns. Some of them were nominal columns (e.g., "name1", "name2") but the majority turned out to be numerical values in quotes (e.g., "5", "100"), quoted ranges (e.g., "1-5", "50-100") or a dictionary with key-value pairs in quotes. For the quoted numerical values, I simply stripped the quotes and converted them to numbers. For the quoted ranges, it was not straightforward because these columns have uncertainty information in them. For instance, "1-5" means any values from 1 to 5 and there was no way to know which value was chosen in the end and also what type of probability distribution was used for the random draw. Although I initially considered treating them as nominal, I decided to take the mean of the minimum and maximum values of a range value: for "1-5", it would be represented as (1+5)/2 = 3. This way, it might be possible to preserve some numerical information about the range in the input data. For the quoted dictionary, I parsed them and expanded to multiple columns so that each key became a column in the input dataset. Finally, I dropped columns that were non-informative (i.e., single unique value) and duplicates. This resulted in about 10% increase of the number of columns, which was still in the range of several hundreds. Whenever a change is made to the design, the set of the input settings may change. In this project, on average, less than 5 columns (including 0) were either added or removed every time the tests were run. When building a training dataset from tests across multiple days with different input settings, I used the union to include all. Here, to impute missing values from the settings absent in the past, I used the domain knowledge of the verification engineers. When preprocessing a set of new test candidates for prediction, I dropped the input settings that are absent in the feature set of the pre-trained models. The output datasets did not require preprocessing.

Models

I used an ensemble of a supervised and an unsupervised learning models. Due to the severe class imbalance between passes and failures (near 99% pass and 1% failure rate) in the training data, it is possible to either train a supervised model with adjusted class weight or train an unsupervised model that detects outliers (i.e. failures). For the unsupervised, because the majority of the training data is passed tests, it is possible to consider the failures as outliers or abnormalities. In a preliminary analysis, I found that the supervised and the unsupervised models provided predictions that were qualitatively different; the unique failure signatures (UFS) from the supervised model's and the unsupervised one's predictions were not identical although there were some overlaps. Thus, when the union of both predictions were computed, there was a small increase of UFS recovery across many testing datasets. Hence, I decided to use both models and take the union of the predictions. This means, when test candidates are passed to the model for prediction, a candidate will be flagged as failure either of the supervised or the unsupervised predicts it as failure.

Due to the frequent changes in data generation process, I decided to use algorithms robust to frequent retraining and tuning

Model candidates	Recall	Efficiency
#1	0.70	1.25
#2 (chosen)	0.66	1.85
#3	0.85	0.55
#4	0.25	2.50

TABLE 1: *Example of model candidate scores and how the best model is chosen. In the tuning process, both recall and efficiency are considered. Efficiency of 1 means the ML flow is as efficient as the random flow. This becomes the lower bound of model performance. #3 is ruled out because even though it has the highest recall, the efficiency is lower than 1 (baseline). Then, #1 is the model with the highest recall. However, instead of choosing this, I look at other candidates within a margin (0.05 in this case) from the maximum value of the recall, meaning all the candidates that have recall values between 0.70 (maximum) and 0.65 (=0.70-0.05). In this example, #2 has higher efficiency than #1 and is within the recall margin. Hence, #2 is chosen as the best model.*

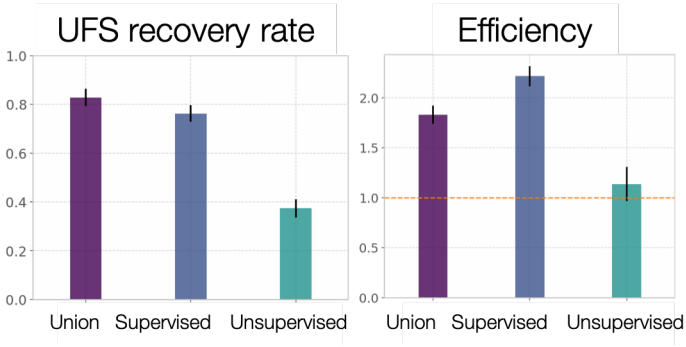


Fig. 3: *The unique failure signature (UFS) recovery rate (left) and efficiency (right) metrics across 15-day (1 month, the tests were not generated daily during this duration) performance for the three models (union, supervised and unsupervised). The dashed orange line in the efficiency plot shows average fail-discovery rate (the lower bound of the efficiency metric). Note that the union approach catches more UFS but lowers efficiency because more tests should be run.*

(i.e., faster training time). I used a group of non-neural-net scikit-learn (v0.20.2) classifiers as supervised and isolation forest as unsupervised learning algorithms. For both cases, I conducted randomized search to tune the hyperparameters and select the best model. For the supervised, I used algorithms such as logistic regression and tree-based ensemble methods (random forest, gradient boosting, and extra trees). The winning algorithm was the logistic regression with L2 regularization, potentially because the preprocessed input data had high sparsity (i.e., more than 50% was 0 after imputation).

Engineers care more about the unique failure signatures than simple binary labels. When a number of failures are found in test simulation, if the majority have the same failure signatures, it means engineers found failures that are very similar to each other, which has little value to them; the ultimate goal of verification is to find every possible type of failure (bug) to make the design bug-free. The more UFS engineers find, the more likely to find novel failures. Moreover, if we find more UFS by running fewer tests, it bring higher efficiency in the procedure. Hence, it would make sense to have an objective function that maximizes the number of UFS found, for instance, by formulating the problem as multi-class classification where each class corresponds to a failure signature. In the training data, each failure signature is found mostly just once or a few times, which makes it difficult to use in model training. However, I found that the number of failure signatures increases with the number of failures (Fig. 2); the more failures we find, the more unique failure signatures are retrieved. This suggests that as long as the binary approach works well and catch more failures, it will be natural to retrieve more unique failure signatures.

Metrics and hyperparameter tuning

For both supervised and unsupervised models, I used recall and precision as basic metrics (for model selection in the tuning process). In general, it is not easy to evaluate unsupervised models but in this case, I have labeled datasets and hence it was possible to use the classification metrics. I also used more practical metrics to increase interpretability and address unique failure signatures, which engineers care about. I defined the following two metrics: *unique failure signature (UFS) recovery rate* and *efficiency*.

$$\text{UFS recovery rate}^{1} = \frac{\text{card}(S_{\hat{y}=1})}{\text{card}(S_{y=1})},$$

where S is a set of UFS, y and $\hat{y}$ are true and predicted labels of failure (0 as pass and 1 as failure), and $\text{card}(S)$ is the cardinality of the set S, also known as the unique count of the set. Hence, $\text{card}(S_{\hat{y}=1})$ means the number of the UFS in the tests that are predicted as failure and $\text{card}(S_{y=1})$ as the total number of UFS in all failed tests in training data. This metric is similar to recall but here the focus is on the retrieval of UFS instead of the binary labels.

$$\text{Efficiency} = \frac{\text{Precision}}{\frac{\sum_{i=1}^{N} y_i}{N}},$$

where N is the total number of the tests in the training data. In the deployment setting where both the default and ML flows exist, N is the total number of the tests in the *default* flow. The efficiency metric is defined to easily understand how efficient the ML flow is compared to the baseline (the random flow). The numerator is the precision of the ML flow. The denominator is the proportion of the failures in training data (or the tests in the random flow), which means how often failures are found on average when running randomized tests (i.e., average fail-discovery rate). This metric can be used as a lower bound of model performance. Since engineers want to discover as many failures as possible, this would mean maximizing recall. Due to the trade-off between recall and precision, this attempt would decrease precision. However, the precision should not be lower than the average fail-discovery rate, because otherwise, the random flow would be enough or even more efficient than the ML flow at finding failures. Therefore, desired model performance should show the efficiency score larger than 1.

Since the efficiency metric provides a lower bound to model performance, when tuning the hyperparameters, instead of looking at the combination with best recall, I use the following rule to select the best model. First, the model candidates with the

1. This metric is the same as the Jaccard similarity of $S_{\hat{y}=1}$ and $S_{y=1}$. When Jaccard similarity is used as a metric between two arbitrary sets A and B, it is often assumed that $|A-B|$ and $|B-A|$ are non-zero (i.e., $A \not\subset B$ and $B \not\subset A$). In this case, $S_{\hat{y}=1} \subset S_{y=1}$, and thus I defined the UFS recovery rate with set cardinality.

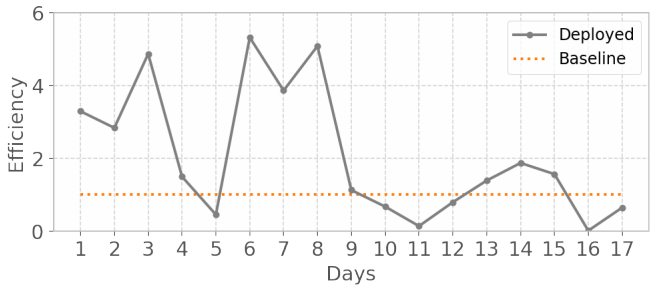


Fig. 4: *First 17 days (3k-4k tests per day) of model performance (efficiency) after deployment. The performance fluctuates widely (all the way up to more than 5 then sometimes plummet to zero). Note that the models have not been retrained during this period.*

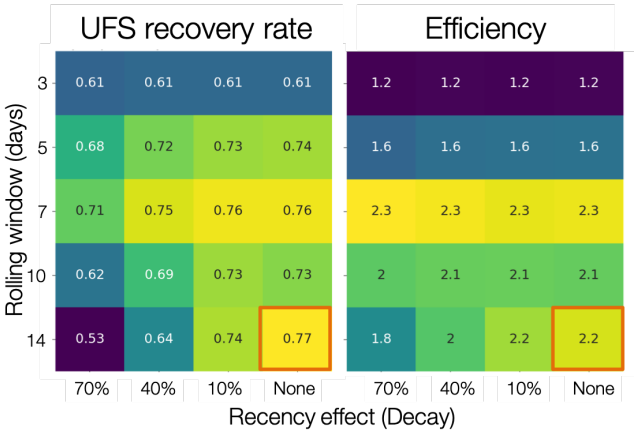


Fig. 5: *Average model performance metrics obtained by simulating retraining scenarios for the training data size (rolling window) and the relative importance of recent data (weight decay). The x axis shows decay parameter, which decides the weights applied to training data. The larger the weights, the faster the decay, meaning old tests become much less important. The y axis shows the rolling window size as the number of days. This decides the training data size; 10 means the training data consists of the tests gathered for the past 10-days. For both plots, brighter colors indicate more desirable results. The marked orange squares show the final decision on training (i.e., 14-day window without decay)*

efficiency score smaller than 1 are dropped because they are less efficient than the baseline. Next, the maximum of the recall values from the rest of the candidates is identified. Instead of selecting the candidate with the maximum recall, I set up a recall margin (0.05) from the maximum recall and check whether there are candidates that are within the margin. Consider this as looking at not just a single model with the best recall but multiple models with close-to the best recall values. Among these candidates, I chose the one with the highest efficiency. This way, without compromising the recall too much, the model with higher efficiency can be chosen. The example is shown in Tab. 1.

Results

For the *snapshot* dataset, the testing data (50% holdout data in 10 different sets; each set is generated independently) shows that the union predictions from the trained supervised and unsupervised models achieved a UFS recovery rate of 82 ± 2 % (mean $\pm$ sem) and an efficiency of 1.8 ± 0.1 (mean $\pm$ sem). Similar results were obtained in the *1-month* dataset (Fig. 3). Note that in the figure, the UFS recovery rate increased for the union approach but the efficiency was sacrificed because the union approach naturally required running more tests. Since the precision score was very low (due to the class imbalance), I ran a permutation test as a sanity check (100 runs) and found the model performance was significantly different from the permuted runs ($p = 0.010$ for the *snapshot* dataset). Overall, in both datasets, on average, the union approach flagged about 40% of the tests and was able to retrieve 80% of the unique failure signatures. This suggests that with the ML flow, it is possible to find 80% of UFS by running only 40% tests, compared to the random flow (baseline).

Post-deployment analysis

Deployment

Several productions engineers and I wrote Python and shell scripts to build a command-line tool that verification engineers can run without changing their main random flow. The script takes test candidates as input and makes a binary prediction on a test candidate's success (pass or failure) based on the pre-trained models (both the supervised and the unsupervised and then their union). Whenever new test candidates are provided to the tool, the input settings of those are preprocessed so that they are consistent with the training data. The tool is provided with 1k test candidates, generated from the testbench, and it flags about 400 tests on average. The number of test candidates provided depends on the computational resources available in the internal cluster. In the flagging (i.e., prediction) process, the script returns the unique identifier of the flagged test candidates. Then it invokes a testbench simulation where only the filtered tests are run. The scripts are deployed as a part of the production team's continuous integration.

After the deployment, model performance started showing high variability, sometimes very different from the pre-deployment model performance. Figure 4 shows the model performance of the first 17 days of post-deployment period. Note that the models were not retrained during this period. During this period, the efficiency scores were often larger than one but they changed dramatically sometimes. In the following sections, I will discuss how to identify the cause of the performance variability in the context of model retraining, and other issues found during the post-deployment stage.

Data for retraining

During the initial period of post-deployment, the models were manually retrained whenever major changes were made either in the tool or in the design. To automate the retraining process, I tested ideas related to the model retraining. First, for any retraining, the size of training data should be determined. Technically, it is possible to use the entire historic data from the very beginning of the testing process. However, this is not a good idea because the training data will be too big and very old tests would be useless since the design would have changed a lot since then. To determine how much training data is needed, I conducted an experiment by considering these two factors: rolling window size and weight decay. The rolling window size corresponds to N consecutive days ($N = 3, 5, 7, 10, 14$) to look back to build a training dataset. For instance, if $N = 7$, tests that were run for the past 7 days become the training data.

The weight decay is related to how *fresh* the data is. If tests were generated more recently, they might be more important

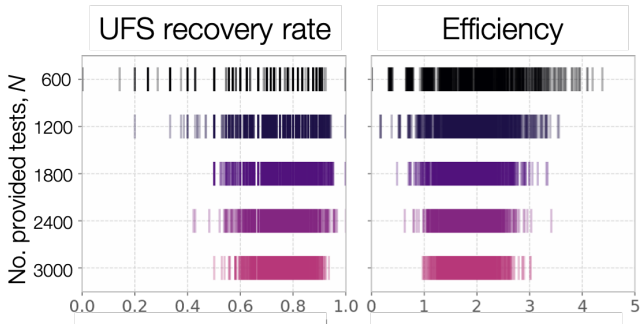


Fig. 6: *The effect of the number of tests that are provided to the models and the performance variability. Each vertical line in the raster plots represents a single simulated run. The model performance is more variable when fewer tests are provided to the models. It may imply that the performance depends on the quality of the test candidates, which can vary more if the number of the test candidates provided is smaller. The more tests we provide, the less variable the performance becomes.*

because the design then is more similar to the current day's compared to older tests. The multiplicative power decay is used to compute the weight w, ($w(t) = x^t$, where x is the power parameter (0.3, 0.6, 0.9, 1 (=no decay)) and t is the number of days counting from today). Using the power law, $x = 0.9$ would mean tests from yesterday are 10% less important than today's. Once the weights are computed, they are applied to the objective function during training by using `sample_weight` parameter in scikit-learn models' `fit()` module. It allows users to assign weights during model fitting for every data point. Since multiple tests are generated on a day, they each get the same weights and the weights only vary on the day-level. Note that this weight adjustment is added on top of the class weight adjustment (`class_weight='balanced'`).

All combinatorial scenarios between the rolling window and weight decay were tested via simulation across multiple datasets (Fig. 5). When the rolling window was too small (e.g., $N = 3$), performance was low for both the UFS recovery rate and the efficiency, which suggests the 3-day window might be too small to construct a good training dataset. A faster decay (small power parameter) tends to mimic the effect of having a smaller rolling window and generally degraded performance. As shown in Fig. 5 as an orange box in each grid, the final decision was to have a 14-day window without any decay even though the efficiency value was slightly higher in the 7-day without any decay. This was to consider the fact that it is possible to run a smaller number of tests in the future due to the potential cluster resource constraints and thus the 7-day window might not provide enough tests for training.

Random-draw effect

It is suspected that the fluctuation in model performance (Fig. 4) might be related to the quality of the test candidates. This is because the test candidates were generated randomly in the testbench independently and we have no control over it. Hence, by chance, it is possible that the test candidates on a certain day might be more challenging to the models (e.g., samples that are closer to the decision boundary), which may result in low performance (i.e., "random-draw" effect). To test this idea, I simulated the effect of the random draw by varying the number of test candidates provided to the models (Fig. 6). I found that when more candidates were provided, model performance was more stable for both UFS recovery rate and efficiency. In the actual deployment, about 1000 test candidates were provided to the tool. As shown in Fig. 6, it is very much possible that with 1000 candidates, the efficiency can be lower than one or as high as four in certain draws. For the simulation in Fig. 6, I drew tests from a pool of 250k tests but considering that the actual number of possible test candidates that can be ever generated is astronomical, variability in model performance due to the random-draw effect could be more severe in reality.

Top-K approach with periodic retraining

Although the predictions from the supervised and the unsupervised models are binary in the deployed tool, in fact both models (logistic regression and isolation forest) can return a continuous score, which can be used as a measure of likelihood of failure. For the supervised model, this is prediction probability and for the unsupervised, this is anomaly score. In the default setting (as in the deployed tool), the supervised model classifies the candidates with the probability of failure larger than 0.5 as failures, and the unsupervised flags the ones with negative anomaly scores as outliers.

To address the random-draw effect, it might be better to use these likelihood metrics. With these metrics, the test candidates can be ranked and the tool can choose the top candidates, which are more likely to fail (prediction probability for a supervised model) or more abnormal (anomaly score of an unsupervised model) than other candidates. Then it is possible to provide a larger number of test candidates to the models, which can simply choose the top K candidates. This allows the models to see more test candidates, which can potentially reduce the random-draw effect. It also works well with the deployed tool because the test candidate generation is very fast and doesn't cost much. Assuming that enough test candidates are provided to the models, it is not necessary to set specific cut-offs for the likelihood measures but to pick the top K tests where K will simply depend on the cluster resource constraints, which is more straightforward.

To test the idea of the top K approach, I ran simulations using the tests collected during the post-deployment period, retrospectively (cf. note that tests were not run every day) (Fig. 7) and also simulated model retraining. I set $K = 400$, then simulated and compared the following three scenarios:

- *Random K*: K tests that were randomly drawn from the tests that were run in the existing random flow. Approximately, 3k-4k tests were run daily and thus, a subset of K tests ($K = 400$) were randomly drawn and this process was repeated multiple times. Note that this flow does not involve the ML models. This simulation is to approximate the average outcome of the random flow when K tests are run. It is represented as gray dot-line (mean and sem from 100 random simulations) in Fig. 7.
- *Top K without any retraining*: Top K candidates flagged as failure by the models. The models saw the input settings of the *tests that were run in the existing random flow* (the same 3k-4k tests from the *Random K*). Using the same tests as in the "Random K" is important to make the comparison fair and consistent. In this scenario, both supervised and unsupervised models were never retrained. This is shown as blue dots in Fig. 7.
- *Top K with retraining under "three-strikes" rule*: Same as the previous but both models were retrained whenever

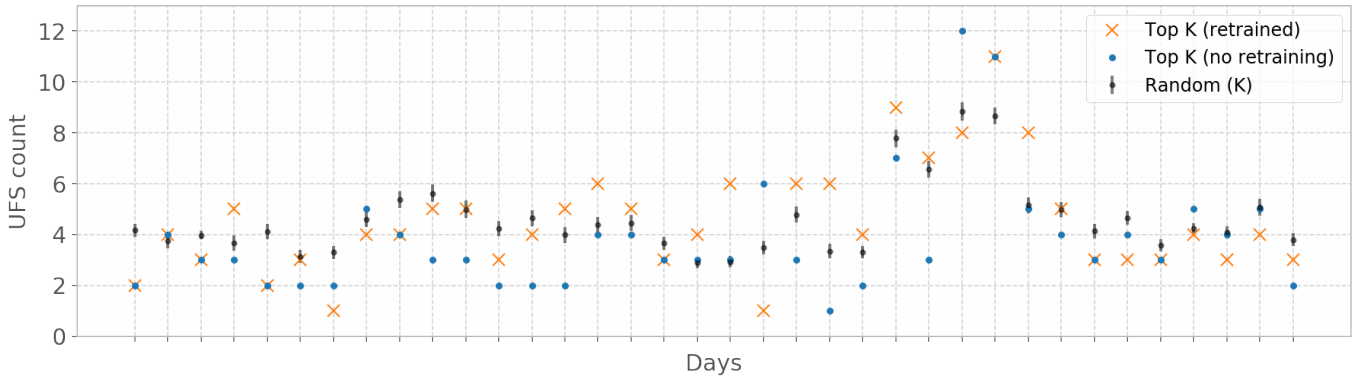


Fig. 7: *Comparison in UFS* counts *between the randomly drawn K tests and the model-filtered K tests (K=400) for 36 days after deployment. The prediction probability and the anomaly score were used to rank the filtered test candidates and choose the top K tests to run (the orange crosses and blue dots), for the supervised and the unsupervised model, respectively. For the orange crosses, the models were retrained and tuned whenever the model performance was worse than the baseline, three days in a row. The blue dots show the scenario without any retraining throughout the whole period. The gray dot-line plot shows mean and 95% confidence interval of performance generated from 100 random draws from a pool of 3k tests from the random flow (daily). Since all scenarios that are compared here have the same number of tests, direct comparison of the UFS counts is available.*

model performance was lower than the *Random K*'s, three days in a row. It is shown as orange crosses in Fig. 7.

Since all scenarios have the same $K = 400$, it is possible to compare the unique failure signature (UFS) counts (the y axis in Fig. 7) instead of the UFS recovery rate. Although the models did not always perform better than the baseline, when they did (the middle section of the figure), retraining the models based on the "three-strikes rule" did help. This rule kept the models relatively new but also helped keeping good models without retraining too frequently. In the middle section of the figure, it was possible to use the same models without retraining for almost two weeks. Theoretically, it is possible to retrain the models every day. However, model retraining is not free and it still consumes computational resources in the internal cluster. This means, too frequent retraining can undermine the benefit of using the ML models.

This simulation was based on the 3k-4k tests that were run almost daily. To compare the model performance and the random-testing results, it was important to use the same set of tests for the simulation; the models saw the input settings of the same 3k-4k tests and made predictions which were then compared to the actual results. In this case, the models have seen only 3k-4k of test candidates but in reality, if the top K approach is adopted, it will be possible to increase the number of candidates provided to the models, which may potentially increase model performance given that the models see a larger number of the candidates. During the mid two-week period in Fig. 7, on average, the "top K with retraining" approach was able to obtain 2.62 ± 1.21 (mean $\pm$ std) more UFS compared to the random flow. According to verification engineers, even a single additional UFS is valuable once the design is mature and failure rate is low. Hence, if the top K approach is applied with a larger number of test candidates, it will be possible to find even more UFS.

Opportunities for enhancement

This project still has room for improvement in terms of data and modeling. From the data perspective, first, it's necessary to gather more information on data drift to easily debug abnormal model performance. Aside from the above-mentioned random-draw effect, the main culprit of the decrease in model performance is a change in the design or the testbench. Currently, it is difficult to understand a change with immediacy and to measure its degree. For instance, in Fig. 7, model performance was worse than the baseline for multiple days and it was very difficult to pinpoint the reason; the only option was to increase the retraining frequency. A possible idea to cope with this problem is to measure a change in the design or the testbench, by comparing commits although this might not necessarily reflect a high-level functional modification. Thus, further discussion with domain experts is needed to find a better solution.

Second, the input dataset quality can be improved by reducing the randomness of the input settings. An important modeling challenge comes from the fact that two identical input settings can result in different outcomes because there is stochasticity in the test generation process. Considering that a testbench cannot be completely deterministic without a design overhaul, exposing subsequent settings controlled by the input settings can provide additional features to the input dataset.

Third, how to measure failure signatures can be improved as well. Currently, the unique failure signatures are 8-digit hexadecimal codes from a hash function based on failure log files. Engineers do not use any similarity metrics between hexadecimal codes and whether a distance between two hexadecimal codes is meaningful is unknown. Instead of using a hash function, it is possible to directly extract semantic information from the failure logs and use it to group and label failures in a meaningful way. This can improve interpretability of the failure signatures and make it possible to build a multi-class classifier, if the log files can be categorized into several groups.

The quality of the union approach depends on the performance of both supervised and unsupervised models. Currently, the vote from each model has the same weight; when one of the models flags a test candidate, the candidate is predicted as failure. However, it is possible that the two models have different performance, meaning one model might have better reliability than the other. In fact, in the early stage of the post-deployment period, I found that the supervised model performance was better than

the unsupervised but in the later stage, it was the opposite. This might have been caused by the design maturation over time, which decreased the number of failures in the training dataset. Therefore, it is possible to consider the difference in model performance when using the ensemble approach. Related to this issue, it is possible to apply different rolling window sizes to the two models. For instance, the supervised model might require a larger training dataset to obtain more failure examples. It would be possible to find the optimal rolling window size for each model by running an experiment similar to Fig. 5.

Conclusions

Hardware verification is a costly process in microprocessor manufacturing, especially when design is mature and failures are rarely found. At this stage, the default randomized testing gets redundant and manual intervention from verification engineers is often required, which is time-consuming. By using the input setting values and test outcomes from the tests that were run previously (99% pass rate), it was possible to train machine learning models that reduce the number of tests to run by 60% while retrieving 80% of unique failure signatures on average. This indicates, engineers can run fewer tests to retrieve similar number of unique failure signatures. Currently, the models have been deployed and used by production engineers to make the verification process more efficient.

In real-world scenarios, it is often the case where a machine learning approach faces many practical constraints. Hardware verification turns out to be a good example. Verification tests are randomly generated and the information about the randomization is intractable, which makes it difficult to control test generation or measure the degree of the stochastic behavior. Also, ML models are only useful in the later stage of hardware verification when it is not easy to find failures by running random tests because in the beginning, the random testing can find a number of failures easily. This means, to use machine learning for the failure detection in hardware verification, one will inevitably face severe class imbalance. Modifying the objective function so that it actually considers the metric of interest, unique failure signature, is not easy because simply there are not enough training examples for each signature. On top of this, the design and even the testbench itself change frequently, suggesting that the data generation process goes through frequent changes.

To address these issues, this study shows a prototype that provides test candidates and filters out failure-prone tests instead of trying to guide the testbench itself. To work with the class imbalance issues, I used both supervised and unsupervised models to address the problem as classification and outlier detection at the same time. I chose a customized approach for model selection by evaluating multiple metrics to be more practical and be able to make a compromise between the metrics. Finally, I have conducted experiments in the post-deployment process to address the details of retraining and identifying the cause of performance variability, which are often overlooked but crucial in deployment. In summary, this study proves that a machine learning approach can be used for failure detection in hardware verification. It also provides an example to work under practical constraints and investigate performance-related issues in building actual machine learning products.

Acknowledgments

I thank Wade Walker in Research, Nagesh Loke, Aneesh Balaji Ganesh Ram, Swati Ramachandran, and Mark Koob in Central Engineering at Arm for their invaluable support and help for data collection, software engineering, prototyping, discussions on metrics and objectives, and domain knowledge on hardware verification. Building a machine learning engineering product requires collaborative effort and I have been very fortunate to work with these great collaborators. I also thank Meghann Agarwal and Dillon Niederhut for reviewing the manuscript and providing useful comments, and the SciPy conference for the opportunity to present this work.

References

[Wil05] Wile, Goss, & Roesner. 2005. Comprehensive functional verification: The complete industry cycle (Systems on silicon), Morgan Kaufmann Publishers Inc. ISBN:0127518037

[Ioa12] Ioannides & Eder. 2012. Coverage-directed test generation automated by machine learning - A review. ACM Trans. Design Autom. Electr. Syst.. DOI:10.1145/2071356.2071363

[Mam16] Mammo, Furia, Bertacco, Mahlke, & Khudia. 2016. BugMD: automatic mismatch diagnosis for bug triaging. In Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference. DOI: 10.1145/2966986.2967010

[Ber13] Bernardeschi, Cassano, Cimino, & Domenici. 2013. GABES: A genetic algorithm based environment for SEU testing in SRAM-FPGAs. Journal of Systems Architecture. 59-10, Part D. DOI: 10.1016/J.SYSARC.2013.10.006

[Cru13] Cruz, Martinez, Fernández, & Lozano. 2013. Automated functional coverage for a digital system based on a binary differential evolution algorithm. Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC). DOI: 10.1109/BRICS-CCI-CBIC.2013.26

[Bar08] Baras, Dorit, Fournier, & Ziv. 2008. Automatic boosting of cross-product coverage using Bayesian networks. Haifa Verification Conference 2008: Hardware and Software: Verification and Testing. DOI: 10.1007/S10009-010-0160-Z

[Wag07] Wagner, Ilya, Bertacco, & Austin. 2007. Microprocessor verification via feedback-adjusted Markov models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 26-6. DOI: 10.1109/TCAD.2006.884494

[Fin09] Fine, Fournier, & Ziv. 2009. Using Bayesian networks and virtual coverage to hit hard-to-reach events. International Journal on Software Tools for Technology Transfer (STTT). 11-4, 291-305. DOI: 10.1007/S10009-009-0119-0

[Sud08] Sudakrishnan, Madhavan, Whitehead, & Renau. 2008. Understanding bug fix patterns in Verilog. Proceedings of the 2008 international working conference on Mining software repositories. 39-42. DOI: 10.1145/1370750.1370761

Expert RF Feature Extraction to Win the Army RCO AI Signal Classification Challenge

Kyle Logue‡*, Esteban Valles‡, Andres Vila‡, Alex Utter‡, Darren Semmen‡, Eugene Grayver‡, Sebastian Olsen‡, Donna Branchevsky‡

Abstract—Automatic modulation classification is a challenging problem with multiple applications including cognitive radio and signals intelligence. Most of the existing efforts to solve this problem are only applicable when the signal to noise ratio (SNR) is high and/or long observations of the signal are available. Recent work has focused on applying shallow and deep machine learning (ML) to this problem. Feature generation, where raw signal information is transformed prior to attempting classification is a key part of this process. A big question that researchers face is whether to let the deep learning system infer the relevant features or build expert features based on expected signal characteristics. In this paper, we present novel signal feature extraction methods for use in signal classification via ML. The deep learning and combined approaches are discussed in a simultaneous publication. Expert features were utilized via ensemble leaning and shallow neural networks to win the Army Rapid Capability Office (RCO) 2018 Signal Classification Challenge. The features include both standard statistical measurements such as variance and kurtosis, as well as measurements tailored for specific waveform families. We discuss the best statistical descriptors along with a ranked list of signal features and discuss individual feature importance. We then demonstrate our implementation of these features and discuss effectiveness in estimating different modulation classes. The methods discussed when combined with deep learning are capable of correctly classifying waveforms at -10 dB SNR with over 63% accuracy and signals at +10 dB SNR with over 95% accuracy from an Army RCO provided training set.

Index Terms—modulation, feature extraction, neural networks, machine learning, decision trees, wireless communication, signals intelligence, feature importance

Introduction

All conventional communications systems are designed with the assumption that the transmitter and receiver are cooperative and have full knowledge of the waveform being exchanged. However, there are scenarios where the receiver does not know what waveform (i.e. modulation, coding, etc.) has been transmitted. Classical examples include cognitive radio network (i.e. a new terminal enters a network and needs to figure out what waveform is being used), and signals intelligence (i.e. interception of adversary's communications). The problem of waveform classifications, or more narrowly, modulation recognition has been studied for decades [ModRec]. Given the implication of SIGINT[1] applications before cognitive radio, much of the work had not been published. Key early work is done by Azzouz & Nandi [Nandi1], [Nandi2], [Azz1], [Azz2].

The fundamental approach taken by most authors has been to find data reduction functions that accentuate the differences between different waveforms. These functions are applied to input samples and decision is made by comparing the values against a set of multi-dimensional thresholds. Determining the threshold values by hand becomes impractical as the number of clusters and/or functions grows. The idea to apply neural networks to help make these decisions has been around for decades [Azz2]. However, it is only recently that our understanding of machine learning combined with enormous increase in computational resources has enabled us to use ML techniques with many data reduction functions against many simultaneous waveforms.

Challenge Description

The Army Rapid Capability Office is seeking innovative approaches to leverage artificial intelligence (AI) to conduct blind radio frequency signal analysis. To this end, they published a labeled modulation classification dataset and created a competition [Army] to properly classify a pair of unlabeled test sets. This paper details the efforts of The Aerospace Corporation's Team Platypus to build a modulation classification system via *traditional* expert features and shallow machine learning classifiers. In this context, shallow refers to the fact that the ML classifier will not build features out of the raw data, instead the classifier will only use the expert features provided. The winning submission from Team Platypus utilized a combination of this expert feature engineering with a deep neural network trained on raw IQ[2] samples, which are described in a simultaneous companion publication.

The training dataset [Mitre] consists of 4.32 million signals each of which containing 1024 complex (IQ) points and a label indicating the modulation type and SNR. Modulation type is selected from one of 24 digital and analog modulations (including a noise class), with AWGN at six different signal-to-noise ratios (-10, -6, -2, +2, +6, or +10 dB). The complete dataset included 30,000 rows for each modulation and SNR configuration. Sample rate is selected from a set (200, 500, 1000, or 2000 ksps), and symbol rate is selected from a set (4, 8, 16, or 32 samples per symbol). Neither of the rate parameters is included in the label.

* Corresponding author: kyle.logue@aero.org
‡ The Aerospace Corporation

1. Signals Intelligence
2. In-Phase & Quadrature

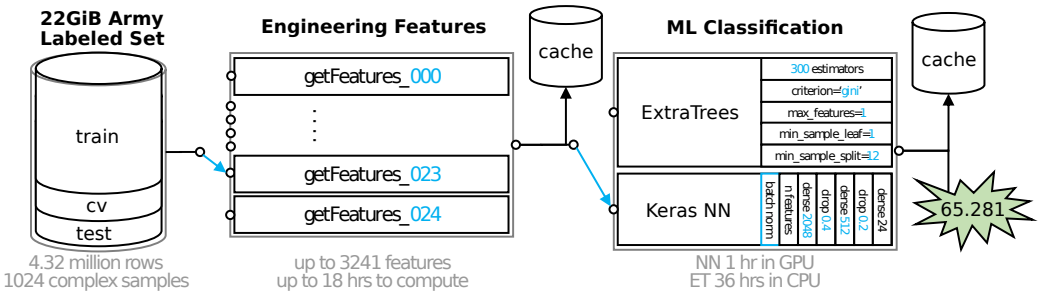


Fig. 1: *Data flow through engineering features evaluation to classification and scoring. Light-blue denote the many variable parameters available. In the Army dataset, cv is short for cross validation.*

The competition consisted of assigning a likelihood score to each of the 24 possible modulation classes for each of the 100,000 rows in a pair of unlabeled test sets.

Classifier performance is evaluated via a pre-defined equation based on the well-known log loss metric, sometimes referred to as cross-entropy loss. The traditional cross validation log loss equation is:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij} \tag{1}$$

Where N is the number of instances in the test set, M is the number of modulation class labels (24), y_{ij} is 1 if test instance i belongs to class j and 0 otherwise, p_{ij} is the predicted probability that observation i belongs in class j. Per [Mitre] this is then scaled between 0 and 100.

$$score = \frac{100}{1+logloss} \tag{2}$$

Note:

- A uniform probability estimate would yield a score of 23.935, not zero.
- To get a perfect 100 score participants would need to be both 100% correct and 100% confident of every estimation.

We will also use a more standard F_1 metric for each modulation is used. This is an excellent measurement of classifier performance since it uses both recall r and precision p, which better account for true positives and false positives:

$$r = \frac{\sum true\ positive}{\sum false\ negative + \sum true\ positive} \tag{3}$$

$$p = \frac{\sum true\ positive}{\sum false\ positive + \sum true\ positive} \tag{4}$$

$$F_1 = \frac{2}{\frac{1}{r}+\frac{1}{p}} \tag{5}$$

Approach

Team Platypus' approach to solve this modulation classification problem is to combine deep neural networks and a shallow learning classifiers leveraging custom engineering features. Both of these are supervised machine learning systems. The engineering features that we applied to this data set are based on traditional signal processing and digital communication techniques. Some shallow learning classifiers, such as Extremely Randomized Trees (ERT) [ModRec] and Random Forests [Nandi1], are decision-tree ensemble methods designed to be robust to overfitting. Ensemble methods train multiple classifiers that will ultimately decide the class using a majority vote or similar metric. These constituent classifiers learn to be different by using different training datasets and/or random parameters independent of the output. The majority voting over this diverse set tends to mitigate the possible overfitting of the constituent classifiers. This is a highly desirable property that becomes even more useful in applications where the test data may have some deviations compared to the labeled train data. The other advantage of decision-tree ensemble methods is that they provide an estimate on whether the features are useful in the classification process. This is further described in Feature Importance Evaluation.

Figure 1 shows the general flow of data through the engineering features evaluation system. The labeled training data is split into training, cross-validation, and testing using a 70%-15%-15% split. When using neural networks, the cross-validation set is the only fair method to prevent network overfitting. When using ERT, the 15% allocated to cross-validation is appended to the training set. Using the Army RCO score metric, the final version of this system scored 65.281. This equates to a cross-validation log loss of 0.532. The output of each step is written to large cache files to enable quick evaluation of new features and integration into the next processing pipeline.

Not pictured are the later steps that merge these expert features with the ResNeXt convolutional deep neural network and a temperature calibration step; all of which yielded an internal final score of 76.422, which equates to a final cross-validation log loss of 0.308.

Measurement Vectors

Multiple transformations of the raw complex measurement vectors were made as intermediate steps to feature extraction. Most of the reduction functions (i.e. feature extraction) are applied to each of the transformed vectors. The following sections describe these methods.

I. Brute-Force PSK & QAM Symbol Estimation

Many common modulations can be expressed in the following form:

$$z_{(t)} = \sum_{n=0}^{\infty} x_{[n]} \cdot h_{(t-T_0-nT_s)} \tag{6}$$

Where $z_{(t)}$ is the received baseband continuous-time signal, $x_{[n]}$ are the complex-valued data symbols (each selected from some fixed constellation, depending on modulation), T_0 is the time offset of the first symbol, T_S is the symbol period, and $h_{(t)}$ is the pulse-shaping impulse response. This broad description includes all

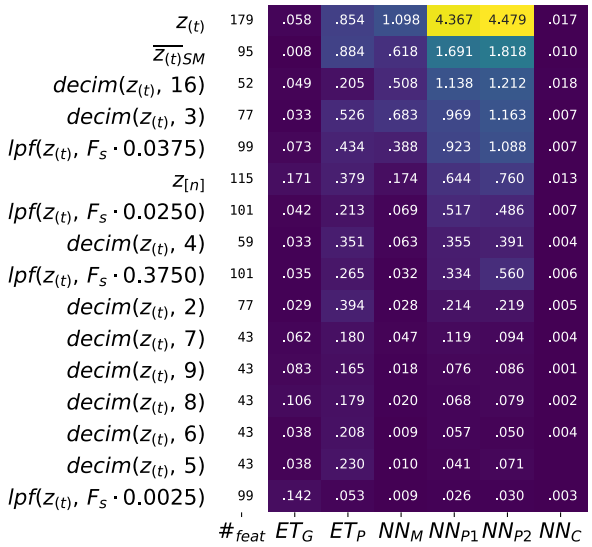


Fig. 2: *Ranked importance of measurement vectors. Numbers in the heat map indicate residual crossvalidation logloss. See Feature Importance Evaluation for a description of the ranking statistics.*

ordinary PSK[3], APSK[4], and QAM[5] modulations, and it can be extended to include variants such as OQPSK[6], $\frac{\pi}{4}$ QPSK, etc.

Given $z_{(t)}$ (or its discrete-time approximation), the blind symbol recovery operation determines T_0, T_S, and $h_{(t)}$ in order to estimate $x_{[n]}$ without attempting to determine the precise modulation type.

For the Army RCO Challenge, this process is greatly simplified because T_S may only take one of four discrete values: 4, 8, 16, or 32 samples per symbol. Similarly, $h_{(t)}$ is always the simple rectangular pulse or a root-raised-cosine (SRRC) filter with one of a few rolloff parameters. We simply attempt recovery for all possible combinations of these parameters, estimate SNR using the M_2M_4 method [Pauluzzi], and keep the configuration with the highest SNR. (Note the generic, constant-envelope M_2M_4 method will return biased results for APSK and QAM modulations, but the max-SNR point is still accurate enough for timing estimation.) The pulse-shaping library can be simplified by pre-calculating discrete filter responses for $T_S = 4$, and decimating all other inputs to match that effective sampling rate.

One notable special case is OQPSK. Since the dataset has neither phase nor frequency offsets, this signal can be trivially "converted" to QPSK by delaying the real-part of the input signal by $\frac{T_S}{2}$. This method would not work for real-world signals, but is adequate for the Challenge.

The only remaining parameter is T_0, which we estimated using one of two methods. The first is Seung Joon Lee's "absolute value nonlinearity" method [Lee]. The second is simple brute-force search with a step size of 1/16th of the symbol period, retaining the output with the highest SNR (as above). The former method is selected because it ran considerably faster and returned essentially identical results.

Given all input parameters, we decimate $z_{(t)}$ to four samples per symbol, optionally delay the in-phase part of the signal (see above), apply the selected matched filter, then finally estimate $x_{[n]}$ by applying piecewise quadratic interpolation to the filtered signal.

The resulting symbol set is not used directly, but is used to calculate various statistics (such as the decision-directed noise power) that are used as machine-learning features.

3. Phase Shift Keying
4. Amplitude and Phase Shift Keying
5. Quadrature Amplitude Modulation
6. Offset Quadrature Phase Shift Keying

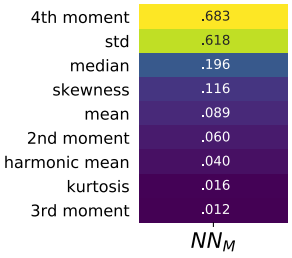


Fig. 3: *Ranked importance of descriptive statistics. See Feature Importance Evaluation for a description of the NN_M statistic.*

II. Phase Histogram

The purpose of this metric is to estimate how many different modulated phases were present in each waveform. The goal is to provide a way to differentiate between different M-ary PSK waveforms.

To this end, we first calculate the instantaneous phase of each input signal $\angle z_{(t)}$. Then divide the interval from 0 to 2π into 32 equal-size bins and count the number of samples within each bin. The resulting histogram is circular-shifted such that the largest count is in first bin. The output feature set is simply the vector of 32 counts, one per bin. Since the input vector size is fixed at 1024 samples, no further normalization is required.

Descriptive Statistics

Descriptive statistics were applied to all vector measurands and accounted for 37% of all engineering features in the most expansive feature functions. Figure 3 details which were of most importance. Note that some of these features are nonlinear combinations of each other.

Custom Features

I. Decision-Directed Noise Estimation

Decision-directed noise estimation operates on recovered symbols. Given a fixed constellation, the estimated noise for each symbol $x[n]$ is simply the difference vector to the nearest constellation point. This nearest-neighbor calculation can be run quickly using k-d trees. The estimated noise power for each constellation is simply the mean-square power of these difference vectors.

Normally, this process would require gain and phase estimation, to correctly align the received signal with the reference constellation. For the Challenge, all input signals had a fixed gain and no phase or frequency offset, so this step is not required.

The estimated noise is calculated separately for a constellation from each of the following modulation types: BPSK[7], QPSK[8], 8PSK[9], 16PSK, 16APSK, 32APSK, 16QAM[10], 32QAM, and 64QAM. Each such estimate is then used as a machine-learning feature.

7. Binary Phase Shift Keying, each symbol representing 1 bit
8. Quadrature Phase Shift Keying, each symbol representing 2 bits
9. 8, 16, and 32 value PSK represent 3, 4, and 5 bits per symbol
10. Similar to PSK Modulations, 16, 32, and 64 QAM represent 4, 5, and 6 bits per symbol

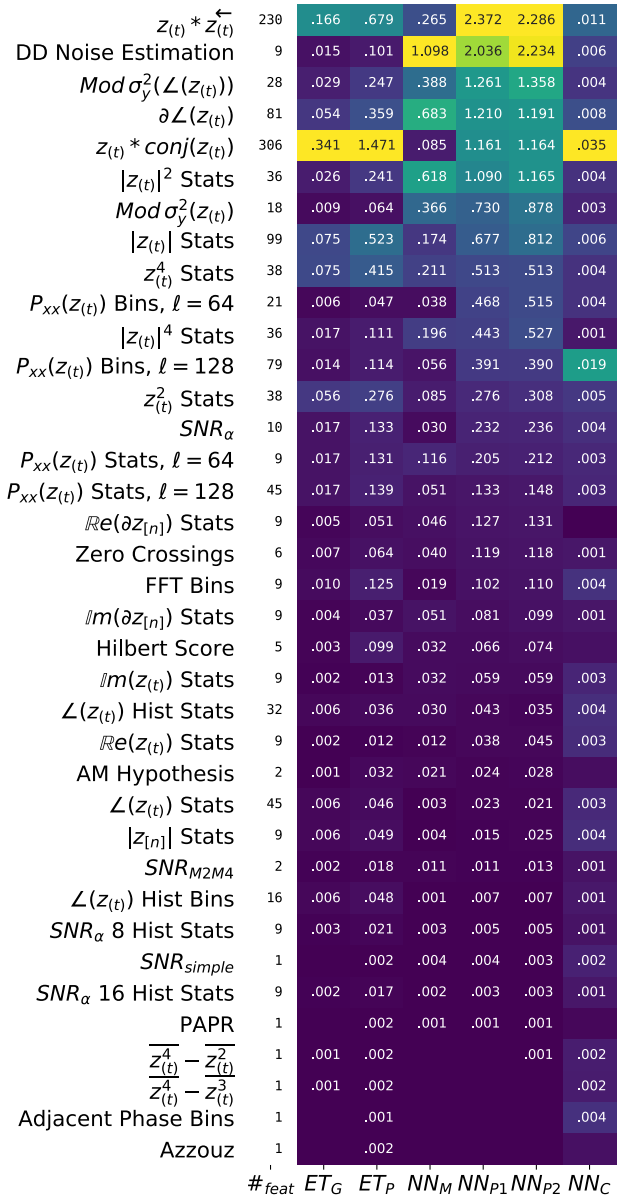


Fig. 4: *Ranked importance of individual features. NN_{P1} differs from NN_{P2} in that these permutaiton importances were derived from two separately trained neural networks. $\#_{feat}$ denotes total number of features in each category noted left. P_{xx} denotes power spectral density. Notice that the color map is normalized per column since metrics are difficult to compare otherwise.*

II. Hilbert Score

An analytic signal is a complex-valued function that has no negative frequency components. The real and imaginary parts of an analytic signal are real-valued functions related to each other by the Hilbert transform. The negative frequency components of the Fourier transform of a real-valued function are superfluous, due to the Hermitian symmetry of such a spectrum. Many techniques for modulating and demodulating single-sideband waveforms use a Hilbert transformer as a core block.

One the most challenging waveforms we had to deal with in this challenge is differentiating between the AM-SSB[11] and AM-DSB[12] pair, especially given the modulation bandwith was as little as 0.5% of the total bandwidth in some cases. The initial intent of this method is to convert time domain data to analytic domain. Another modulation pair that our classifiers had issues with is differentiating QPSK and $\frac{\pi}{4}$ QPSK waveforms. The "Hilbert score" feature is developed to help our classifier reduce confusion among these similar modulations.

The metric is defined as follows:

$$HSM = |\sum H\left(real\left(z[t]\cdot z_0\right)\right)] + \sum H\left(imag\left(z[t]\cdot z_0\right)\right)| \quad (7)$$

Where HSM is the Hilbert score metric, $H(z)$ is the Hilbert transform, z is the vector of input samples, and z_0 is a rotation phasor at either 0 or 45°. This figure of merit proved to be useful to our shallow classification algorithm.

III. DC Power

This metric is simply the 0th bin of the FFT of the complex input vector. The feature consists of the real and imaginary part of this value, considered separately.

IV. Simple SNR Estimation

In principle, given that at the time this metric is implemented we were already using more precise SNR estimators, the usefulness of this simpler and noisier estimator may not have been justified. However, the extremely randomized tree classifier reported this metric as initially useful and we will use it as a baseline for other metrics.

$$SNR_{simple} = \frac{\frac{1}{2N}\sum |z[t]|^2}{Var(|z|)} \quad (8)$$

V. M_2M_4 SNR Estimation

Pauluzzi in [Pauluzzi] presents a comparison of different SNR estimators for phase-shift keyed (PSK) channels with additive white Gaussian noise (AWGN) noise. Though many of those methods are of limited accuracy at very low SNR, the M_2M_4 method still performs well under such conditions.

M_2M_4 method uses the second and fourth moments of a waveform to estimate its SNR. Though it is only directly applicable to constant-envelope signals, it is still useful for relative comparisons under almost any conditions. For simplicity, we use the generalized complex form (m-ary PSK) regardless of modulation:

$$SNR_{M_2M_4} = \frac{\sqrt{2M_2^2 - M_y}}{M_2 - \sqrt{2M_2^2 - M_4}} \quad (9)$$

VI. α SNR Estimation

Many digital communication algorithms require knowledge of the operating signal-to-noise ratio (SNR). Different algorithms exist that estimate signal and noise power or the actual ratio between these two. However, most of the known techniques at low SNR either fail or have very large variance. In order to estimate SNR below 5 dB, we developed a technique that builds on the work by Davenport [Davenport]. This approach to SNR estimation introduces a non-linear technique that uses the inherent properties of non-linear devices, such as a limiter or an automatic-gain-control (AGC) device, to estimate negative SNRs. In our case, the non-linear function used is a sign function. The properties of these devices used for SNR estimation are well known and have been carefully studied in the literature [Davenport]. Similarly to many tracking loops operating at low SNRs, this method multiplies the

11. Single Sideband Amplitude Modulation
12. Dual Sideband Amplitude Modulation

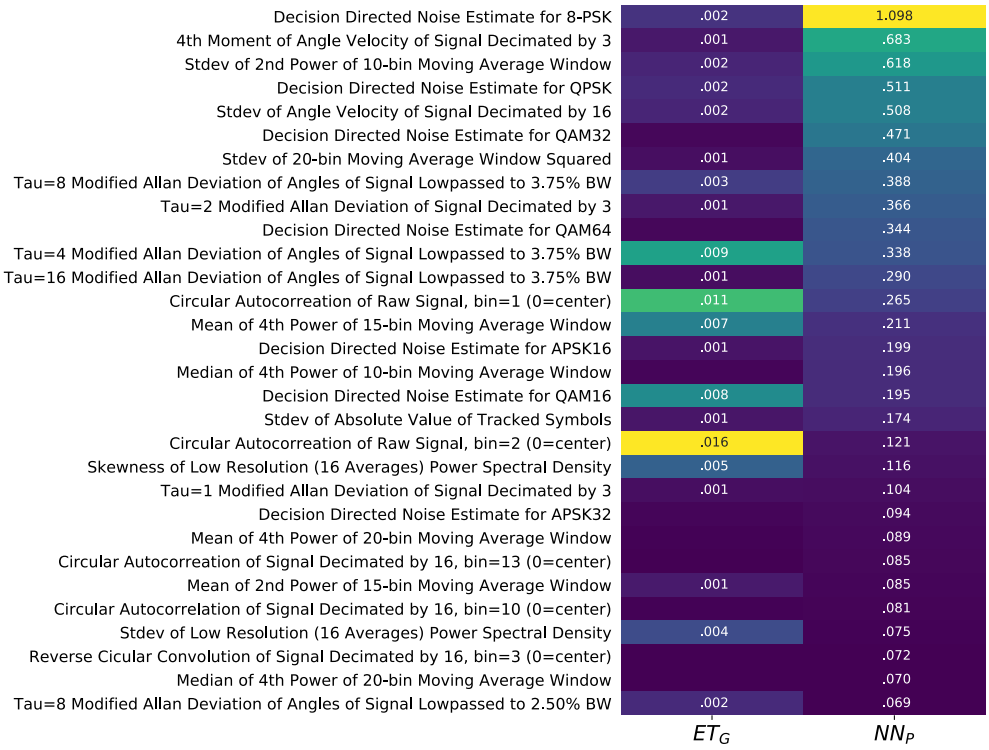


Fig. 5: *Top 30 individual engineering features sorted by neural network permutation importance.*

current sample of a given waveform by the sign of the previous sample (under an assumption of multiple samples per symbol).

$$S_{re}[k] = sign(z_{re}[t] \cdot z_{re}[t-1]))$$

$$S_{im}[k] = sign(z_{im}[t] \cdot z_{im}[t-1]))$$

$$\alpha = \frac{1}{N}\sum sign(S_{re}[t] + S_{im}[t])$$

If the signal is modulated, this process will introduce an error every time the sign of a symbol changes. If the signal has no modulation present, then this block is simply equivalent to a magnitude block. This operation is performed independently on the real and imaginary component of the signal. The metric can be plugged into the result from [Davenport] where for a non-coherent receiver, the SNR can be approximated by:

$$SNR_\alpha = \frac{\alpha^2}{1-\alpha^2} \tag{10}$$

A comparison of the Simple, M_2M_4, and α SNR estimators are shown in Figure 6 and Figure 7.

VII. N-M-D Power Estimation

In the $SNR_{M_2M_4}$ method, we see that the differences of signal moments can be part of the core of SNR estimation algorithms. As we explored generating new features to aid our shallow classifier, we introduced a new feature that would simply compute the difference of two moments $M_x - M_y$. This proved to be of extremely useful as a feature generating function. This function is not meant to compute an approximation metric for estimating SNR but as an intermediate feature in the signal classification process.

VIII. AM Hypothesis Testing

The sample AM signals all were baseband analytic signals with a residual carrier close to zero frequency. The feature we

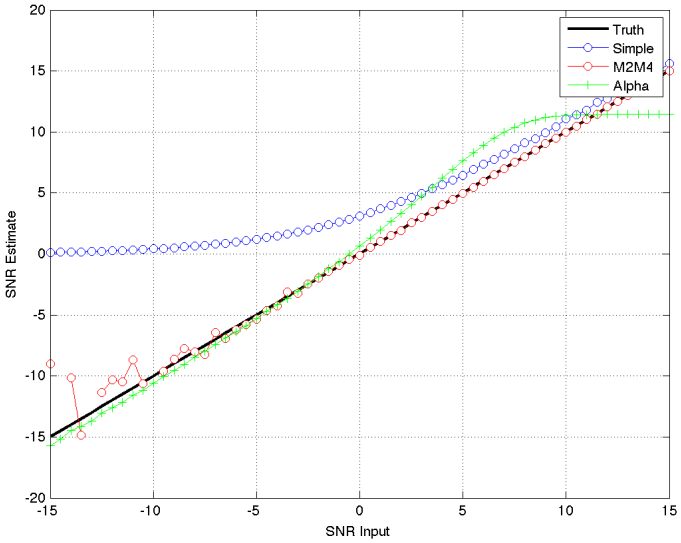


Fig. 6: *Comparison of SNR estimation methods of a PSK modulated signal including novel SNR_α metric.*

designed to distinguish double sideband (DSB) vs. single sideband (SSB) depends on this assumption.

First, the carrier frequency and phase is estimated with the three-sample discrete-Fourier-spectrum interpolator described in section III.D. of Macleod [Macleod]. Multiplication by the inverse of the estimated carrier signal (with unit amplitude) makes the estimated carrier DC. Next, two transformations of the resulting analytic signal are compared.

1. The mean is simply subtracted from the signal: if the signal is DSB, this would result in its coherent demodulation.

2. Non-coherent demodulation is achieved by taking the mod-

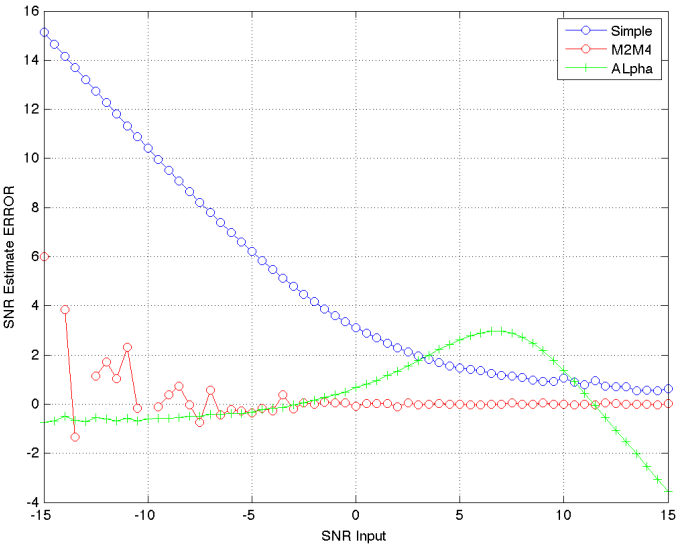


Fig. 7: *Error of SNR estimation methods.*

ulus of the analytic signal and subtracting off its mean over the sample time.

The feature used is the energy of the difference between these two transformations, divided by the energy of the first transformation. When close to zero, the signal would likely be DSB and, when close to one, SSB.

IX. Modified Allan Deviation ($Mod\sigma_y^2(\tau)$)

Typically used as a tool to characterize the stability of time & frequency sources, we applied the modified Allan deviation [NIST] statistic to a number of angle measurements taken of the raw signal and several low-pass transformations. These were computed with a Butterworth 5th-order low pass with cutoff frequencies at 2.5% and 37.5% of the max & min sample rates in order to filter for narrowband modulations.

$$\angle z(t) = arctan2(real(z(t)),\ imag(z(t))) \tag{11}$$

This effectively captured the variability of phase over a number of averaging taus including 1, 2, 4, 8, 16, and 32 complex samples. A nice implementation can be found in the AllanTools[13] python module.

X. Zero Crossings

Some modulations such as $\frac{\pi}{4}$ QPSK are designed such that transitions between symbols avoid passing through the origin. In general, this is used to reduce peak-to-average signal power ratios, which removes certain design constraints on signal amplifiers.

The zero-crossing metric is selected to detect these types of modulations. Considering the real and imaginary parts separately, the metric examines the sign of each sample and counts the total number of transitions from positive to negative or vice versa.

The zero-crossing feature is calculated on the $z(t)$ directly, but is most valuable on the multiple lowpass transformations.

Feature Importance Evaluation

When single or multiple features were added to the feature extraction engine they are computed over all signals in the training set. These features were then appended to the shared cache of features from prior runs. This new larger feature set is then sent to classification and a score is produced.

Initial feature importance is derived from the delta change in score from run to run. This method requires close tracking of every feature and is ambiguous when multiple features or vectors of features are added simultaneously. To address this, several more precise approaches are used to evaluate performance.

A comparison of the following feature importance statistics can be found in Figures 2, 3, 4, and 5.

I. Gini Importance (ET_G)

Gini importance or *mean decrease in impurity*, is implemented in sklearn for Random Forest type classifiers as the *feature_importances_* attribute. After training this metric is available with no additional effort or computation, giving immediate feedback. This metric is useful for the Extra Trees classifier specifically, but is only available for ensemble-type classifiers. While his metric is computationally free, there are several pitfalls described by [Cutler] such as incorrect valuation of correlated or random features that make Gini importance of limited use.

II. Permutation Importance (ET_P & NN_P)

Permutation importance [Parr] can be computed for any classifier by creating a logloss benchmark score for a test set (Eq 1), then randomly permuting 1 feature across all signals. This has the effect of keeping the population statistics of that feature constant, but removing it's contribution to the overall logloss score. Permutation importance is then calculated by subtracting the predicted logloss score of the permuted set from the prior benchmark. Since the shape of the input data is preserved, a trained classifier does not need retraining and is therefor a *fast* metric. We denote permutation importance for ExtraTress and our shallow neural network as ET_P & NN_P respectively. Permutation importance provides the fastest & most robust method for evaluating feature importance for any classifier.

III. Drop-Column Importance (NN_C)

Drop-column importance [Parr] provides perhaps the highest quality estimate of individual feature importance, but is *extremely* computationally expensive and may take weeks or months to compute for even moderately sized neural networks. An initial logloss benchmark is computed, then a feature is dropped across the entire test population, requiring retraining of the classifier for every feature. Resulting importance residuals are difficult to judge since the scale is so small and correlated features often yield near zero change when removed.

IV. Max-Column Importance (NN_M)

Max-column importance is a metric used in Figure 2, 3, and 4, to denote the maximum NN_P across many features grouped into a set. This is computed since the quantiles of importances are heavily skewed toward zero since there are so many (1269) features being compared.

V. Recursive Feature Elimination

Recursive Feature Elimination (RFE) is a technique originally designed for gene selection [Guyon]. This method evaluates the a feature importance estimate of choice after training, then prunes a number of features each step attempting to build a sorted list. This is implemented within *sklearn.feature_selection* as *RFE*. This is

13. https://pypi.org/project/AllanTools/

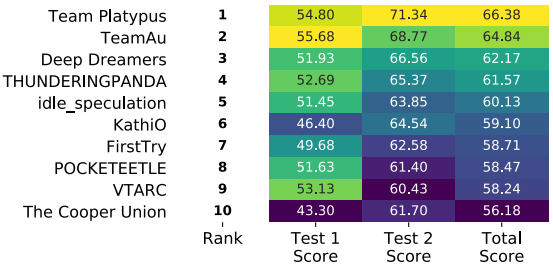


Fig. 8: *Final Army RCO AI Signal Classification leaderboard.*

also *highly* computationally intensive since it requires retraining the classifier every step. There is value in RFE for comparing total number of features to logloss score, especially when building a classifier for low SWAP[14] implementations where computation is limited.

Classification Strategy & Scores

From the beginning of the challenge it was clear that in scenarios where cross validation labeled sets were used to evaluate the performance of classifiers, that ERT have worse overall performance than neural networks. However, given that the nature of the unlabeled sets was unknown, both techniques were pursued.

There were two unlabeled sets released to competitors. Estimates generated for the first set using our deep neural network estimator resulted in very low and inconsistent scores. It was apparent that the data was very unlike the training data initially provided. Team Platypus estimates that only half of the first unlabeled set was like the training set. Only the ERT classifier was applied to that set due to its resiliency to overfitting. Only one of the competitors achieved a higher score (0.8 points) for this set.

The challenge administrators disclosed that the second set contained data 95% like the training set. As such, a combination of a ResNeXt deep convolutional network combined with a shallow two-layer neural network comprised of engineering features was used to submit the winning prediction. Team Platypus held the highest submission score for the duration of the challenge.

Performance

The accuracy of estimation can be visualized as a confusion matrix, shown in Figure 12. Each row represents the true waveform, while each column is the estimated probability. The diagonal values correspond to the 'correct' estimate. Brighter colors indicate higher confidence (e.g. the top left square indicates almost 100% correct identification of the BPSK modulation). This view allows us to quickly identify waveforms that are challenging for our classifier such as the narrowband CPFSK/FSK/FM.

The F_1 score (see Challenge Description) provides another view of the same data. Note that while BPSK is correctly identified 100% of the time, it is not always identified with 100% precision, making the F_1 score less than 1.0. The performance of the classifier decreases at lower SNR. For example, at 10 dB the F_1 score is perfect for most of the waveforms (Figure 10). The overall classifier accuracy versus SNR is shown in Figure 9. Note that we achieve about 50% accuracy even at -10 dB SNR, which is significantly better than previously published results.

14. Size Weight And Power

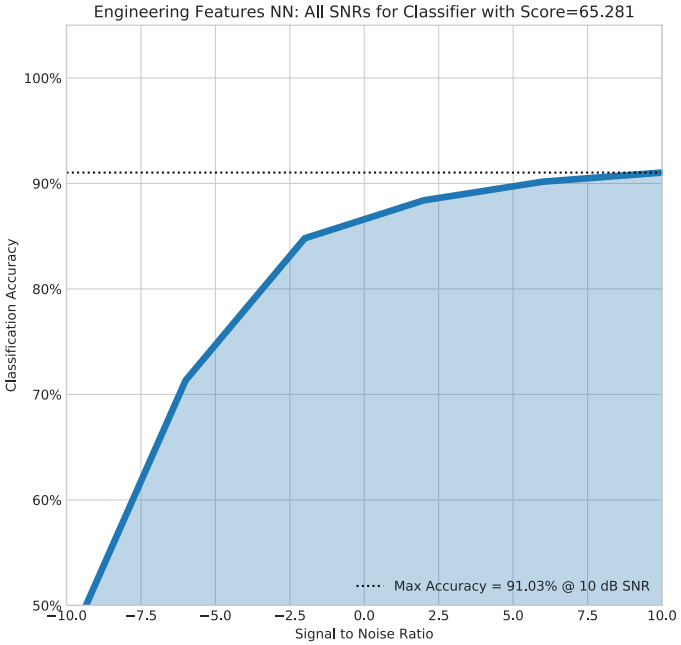


Fig. 9: *Classifier Accuracy vs SNR.*

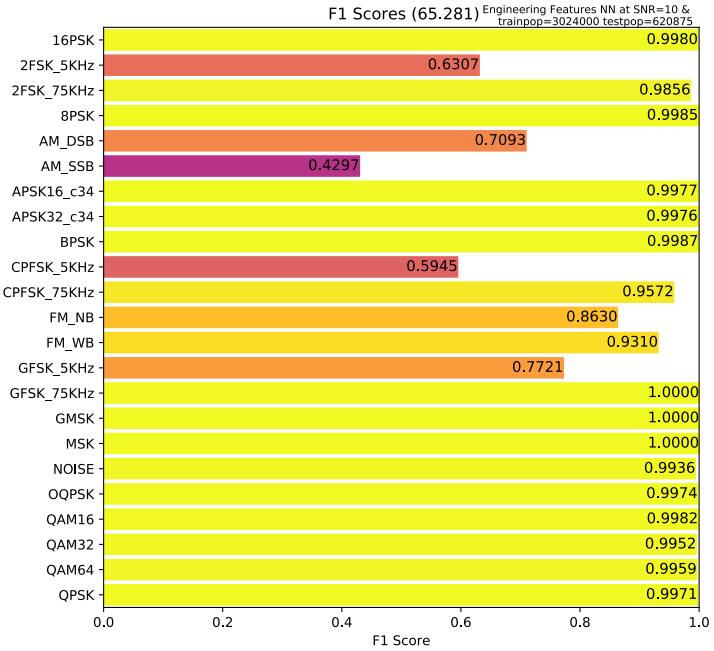


Fig. 10: *F_1 scores at 10 dB SNR signals only.*

Conclusion

The robust results presented in this paper show the significant progress that has been made in application of machine learning over the past decade. However, it is important to note that the test cases offered by the Challenge are somewhat unrealistic. Real-world scenarios would include non-idealities like those found in [OShea].

In regard to feature importance there were a number of interesting results. We emphasize that while Gini importance (ET_G) can approximate neural network permutation importance (NN_P), it can be very misleading when given duplicate or random features. Drop-column importance provides a metric that gives an absolute value of the individual contribution of a feature, but is

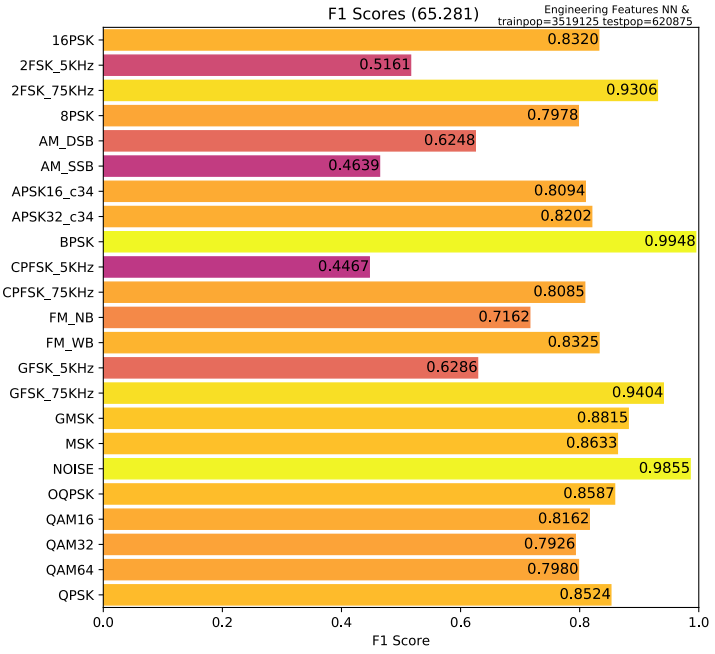


Fig. 11: *F_1 scores for all test data.*

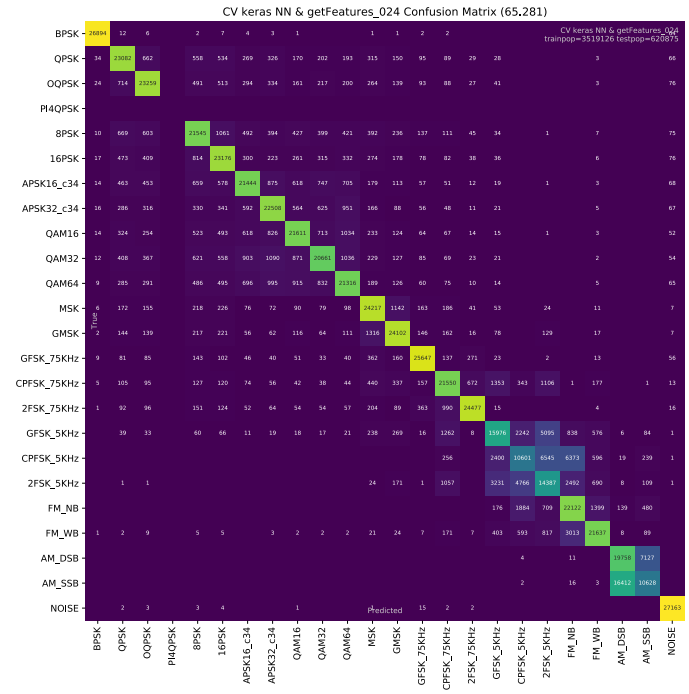


Fig. 12: *Confusion matrix for all test data.*

prohibitively computationally expensive and with correlated features provides almost no value. We generally found permutation importance from our neural networks to be the best measure of feature value in our classifiers, though all methods still generally suffer when features correlate with other features.

We suggest that further research utilize the best statistics and features described herein to achieve modulation classification estimates robust to the traditional pitfalls of deep neural networks, which include generated adversarial networks like those found in [Dong] and [Moosavi] as well as overfitting due to lack of truth data.

Acknowledgements

The authors would like to thank the Army RCO for creating this interesting challenge as well as our competitors who motivated us to stay up late and reconsider our assumptions.

References

[Army] ARMY RCO AI Signal Classification Challenge. (2018). Retrieved from https://www.challenge.gov/challenge/army-signal-classification-challenge/

[Mitre] MITRE Challenge. (2018). Retrieved from https://sites.mitre.org/armychallenge/

[Guyon] Guyon, I., Weston, J., Barnhill, S., & Vapnik, V., “Gene selection for cancer classification using support vector machines”, Mach. Learn., 46(1-3), 389-422, 2002. doi:10.1023/A:1012487302797.

[Pauluzzi] Pauluzzi, D. & Beaulieu, N., “A comparison of SNR estimation techniques for the AWGN channel,” IEEE Trans. on Comm., vol. 48, no. 10, pp. 1681–1691, Oct. 2000. doi:10.1109/26.871393.

[Davenport] Davenport, W., “Signal-to-noise ratios in band-pass limiters,” J. Appl. Phys., vol. 24, no. 6, pp. 720–727, June 1953. doi:10.1063/1.1721365.

[Springett] Springett, J., & Simon, M., “An analysis of the phase coherent-incoherent output of the bandpass limiter,” IEEE Trans. on Comm. Technology, vol. 19, no. 1, pp. 42–49, Feb. 1971. doi:10.1109/tcom.1971.1090611.

[Lee] Lee, Seung Joon. "A new non-data-aided feedforward symbol timing estimator using two samples per symbol." IEEE Communications Letters 6.5 (2002): 205-207. doi:10.1109/4234.1001665.

[Geurts] Geurts, P., Ernst, D. & Wehenkel, L. Mach Learn (2006) 63: 3. doi:10.1007/s10994-006-6226-1.

[NIST] NIST SP 1065: Handbook of Frequency Stability Analysis. 2008. doi:10.6028/nist.sp.1065.

[ModRec] Aisbett, Janet. "Automatic modulation recognition using time domain parameters." Signal Processing 13.3 (1987): 323-328. doi:10.1016/0165-1684(87)90130-7.

[Nandi1] Nandi, Asoke K., and Elsayed Elsayed Azzouz. "Algorithms for automatic modulation recognition of communication signals." IEEE Transactions on communications 46.4 (1998): 431-436. doi:10.1109/26.664294.

[Nandi2] Nandi, A. K., and Elsayed Elsayed Azzouz. "Automatic analogue modulation recognition." Signal processing 46.2 (1995): 211-222. doi:10.1016/0165-1684(95)00083-p.

[Azz1] Azzouz, Elsayed, and Asoke Kumar Nandi. Automatic modulation recognition of communication signals. Springer Science & Business Media, 2013. doi:10.1007/978-1-4757-2469-1.

[Azz2] Azzouz, Elsayed Elsayed, and Asoke Kumar Nandi. "Modulation recognition using artificial neural networks." Automatic Modulation Recognition of Communication Signals. Springer, Boston, MA, 1996. 132-176. doi:10.1007/978-1-4757-2469-1_5.

[Macleod] Macleod, M.D. “Fast Nearly ML Estimation of the Parameters of Real or Complex Single Tones or Resolved Multiple Tones.” IEEE Transactions on Signal Processing 46, no. 1 (1998): 141–148. doi:10.1109/78.651200.

[Cutler] Cutler, A., & Breiman, L. (2018). Random Forests. Retrieved from https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#varimp

[Parr] Parr, T., Turgutlu, K., Csiszar, C., & Howard, J. (2018, March 26). Beware Default Random Forest Importances. Retrieved from https://explained.ai/rf-importance/

[OShea] T. J. O’Shea, T. Roy and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," in IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168-179, Feb. 2018. doi:10.1109/JSTSP.2018.2797022.

[Dong] Dong, Yinpeng, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. “Boosting Adversarial Attacks with Momentum.” 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (June 2018). doi:10.1109/cvpr.2018.00957.

[Moosavi] Moosavi-Dezfooli, Seyed-Mohsen, Alhussein Fawzi, and Pascal Frossard. “DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (June 2016). doi:10.1109/cvpr.2016.282.

Deep and Ensemble Learning to Win the Army RCO AI Signal Classification Challenge

Andres Vila‡*, Donna Branchevsky‡, Kyle Logue‡, Sebastian Olsen‡, Esteban Valles‡, Darren Semmen‡, Alex Utter‡, Eugene Grayver‡

Abstract—Automatic modulation classification is a challenging problem with multiple applications including cognitive radio and signals intelligence. Most of the existing efforts to solve this problem are only applicable when the signal to noise ratio (SNR) is high and/or long observations of the signal are available. Recent work has focused on applying shallow and deep machine learning (ML) to this problem. In this paper, we present an exploration of such deep learning and ensemble learning techniques that was used to win the Army Rapid Capability Office (RCO) 2018 Signal Classification Challenge. An expert feature extraction and shallow learning approach is discussed in a simultaneous publication. We evaluated multiple state-of-the-art deep learning network architectures and adapted them to work in the RF signal domain instead of the image/computer-vision domain. The best deep learning methods were merged with the best expert feature extraction and shallow learning methods using ensemble learning. Finally, the ensemble classifier was calibrated to obtain marginal gains. The methods discussed are capable of correctly classifying waveforms at -10 dB SNR with over 63% accuracy and signals at +10 dB SNR with over 95% accuracy from an Army RCO provided training set.

Index Terms—modulation classification, neural networks, deep learning, machine learning, ensemble learning, wireless communications, signals intelligence, probability calibration

Introduction

All conventional communications systems are designed with the assumption that the transmitter and receiver are cooperative and have full knowledge of the waveform being exchanged. However, there are scenarios where the receiver does not know what waveform (i.e. modulation, coding, etc.) has been transmitted. Classical examples include cognitive radio network (i.e. a new terminal enters a network and needs to figure out what waveform is being used), and signals intelligence (i.e. interception of an adversary's communications). The problem of waveform classifications, or more narrowly, modulation recognition has been studied for decades [Aisbett]. Given the implication of SIGINT[1] applications before cognitive radio, much of the work had not been published. Key early work is done by Azzouz & Nandi [Nandi1], [Nandi2], [Azz1], [Azz2].

The fundamental approach taken by most authors has been to find data reduction functions that accentuate the differences between different waveforms. These functions are applied to input samples and a decision is made by comparing the values against a set of multi-dimensional thresholds. Determining the threshold values by hand becomes impractical as the number of clusters and/or functions grows. The idea to apply neural networks to help make these decisions has been around for decades [Azz2]. However, it is only recently that our understanding of machine learning combined with enormous increase in computational resources has enabled us to use ML techniques to solve this problem.

* *Corresponding author: andres.i.vilacasado@aero.org*
‡ *The Aerospace Corporation*

Challenge Description

The Army Rapid Capability Office is seeking innovative approaches to leverage artificial intelligence (AI) to conduct blind radio frequency signal analysis. To this end, they published a labeled modulation classification dataset and created a competition [Army] to properly classify a pair of unlabeled test sets. This paper details the efforts of The Aerospace Corporation's Team Platypus, the authors of this paper, to build a modulation classification system via deep learning and ensemble learning. In this context, deep refers to the fact that the ML classifier will use the raw IQ[2] data, instead of expertly engineered features. The winning submission from Team Platypus utilized a combination of these deep classifiers and shallow learning classifiers built on expert features which are described in a simultaneous companion publication.

The training dataset [Mitre] consists of 4.32 million signals each of which contain 1024 complex (IQ) points and a label indicating the modulation type and SNR. Modulation type is selected from one of 24 digital and analog modulations (including a noise class), with AWGN at six different signal-to-noise ratios (-10, -6, -2, +2, +6, or +10 dB). The complete dataset included 30,000 rows for each modulation and SNR configuration. Sample rate is selected from a set (200, 500, 1000, or 2000 ksps), and symbol rate is selected from a set (4, 8, 16, or 32 samples per symbol). Neither of the rate parameters is included in the label.

The competition consisted of assigning a likelihood score to each of the 24 possible modulation classes for each of the 100,000 rows in a pair of unlabeled test sets.

Classifier performance is evaluated via a pre-defined equation based on the well-known log loss metric. The traditional log loss equations:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij} \quad (1)$$

1. Signals Intelligence
2. In-Phase & Quadrature

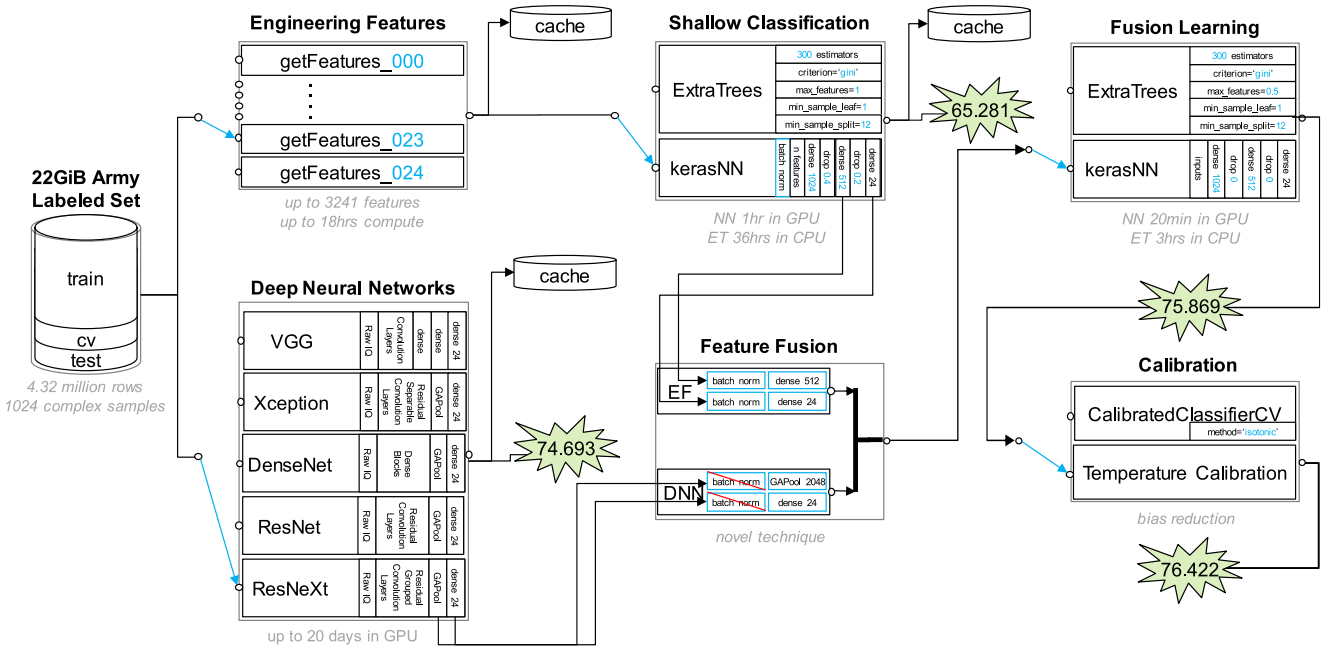


Fig. 1: *Data flow through the classification pipeline. The many variable parameters available are denoted in light blue.*

Where N is the number of instances in the test set, M is the number of modulation class labels (24), y_{ij} is 1 if test instance i belongs to class j and 0 otherwise, p_{ij} is the predicted probability that observation i belongs in class j. The competition score, which we will refer to as simply the score in the remainder of this paper, was defined per [Mitre] as follows:

$$score = \frac{100}{1 + logloss} \tag{2}$$

Notes:

- A uniform probability estimate would yield a score of 23.935, not zero.
- To get a perfect 100 score participants would need to be both 100% correct and 100% confident of every estimation.

We will also use a more standard F_1 metric for each modulation is used. This is an excellent measurement of classifier performance since it uses both recall r and precision p, which better account for false negatives and false positives:

$$r = \frac{\sum true\ positive}{\sum false\ negative + \sum true\ positive} \tag{3}$$

$$p = \frac{\sum true\ positive}{\sum false\ positive + \sum true\ positive} \tag{4}$$

$$F_1 = \frac{2}{\frac{1}{r} + \frac{1}{p}} \tag{5}$$

Approach

Team Platypus' approach to solve this modulation classification problem is to combine deep neural networks and shallow learning classifiers leveraging custom engineering features. Both of these are supervised machine learning systems.

Figure 1 shows the general flow of data through our winning system. The labeled training data is split into training, cross-validation, and testing using a 70%-15%-15% split. When using neural networks, the cross-validation set is used to prevent classifier overfitting. Using the Army RCO score metric, the final version of this system scored 76.422. This equates to a cross-validation log loss of 0.308. The output of each step is written to large cache files to enable quick evaluation of new ideas and integration into the next processing pipeline.

	Rank	Test 1 Score	Test 2 Score	Total Score
Team Platypus	1	54.80	71.34	66.38
TeamAu	2	55.68	68.77	64.84
Deep Dreamers	3	51.93	66.56	62.17
THUNDERINGPANDA	4	52.69	65.37	61.57
idle_speculation	5	51.45	63.85	60.13
KathiO	6	46.40	64.54	59.10
FirstTry	7	49.68	62.58	58.71
POCKETEETLE	8	51.63	61.40	58.47
VTARC	9	53.13	60.43	58.24
The Cooper Union	10	43.30	61.70	56.18

Fig. 2: *Final Army RCO AI Signal Classification leaderboard.*

Classification Strategy & Scores

There were two unlabeled sets released to competitors. Estimates generated for the first set using our deep neural network estimator resulted in very low and inconsistent scores. It was apparent that the data was very unlike the training data initially provided. Team Platypus estimates that only half of the first unlabeled set was like the training set. Our solutions for these datasets relied exclusively on expert engineering feature extraction and shallow classification techniques. Only one of the competitors achieved a higher score (0.8 points) for this set.

The challenge administrators disclosed that the second set contained data 95% like the training set. As such, a combination of a deep learning and shallow learning techniques as described in the rest of this paper was used to generate the submissions for this dataset. Team Platypus held the highest submission score for the duration of the challenge.

Network Type	Best Scores
Simple Convolutional	45.47
VGG	58.38
Modified ResNet	66.21
ResNet-34	72.39
ResNet-50	72.80
ResNeXt-50	74.69
Xception	70.74
DenseNet	65.98

TABLE 1: *Deep Learning Results.*

Deep Learning Modulation Classification

Architecture Search

We implemented multiple Neural Network architectures in Keras using the TensorFlow backend. We begun by testing variations of the networks proposed in [OShea1]. These networks consisted of 2 or 3 convolutional layers followed by 2 or 3 dense layers. We will call these networks "Simple Convolutional". These networks produced scores of around 45 points. We proceeded to test 2 networks proposed in [OShea2], a VGG network and a "Modified ResNet" network. The VGG network produces results around 55 points and the "Modified ResNet" resulted in a score of 59 points.

Our search strategy changed at this point. We conjectured that using the state-of-the-art methods currently applied to image classification would yield good results. Hence, we implemented multiple algorithms by reading their papers and adapting their ideas from 2-dimensional (images) to single dimensional (complex time-series signals). We could not rely on previously built Keras application models since they were all built for the 2-dimensional images classification problem.

We implemented multiple ResNets [ResNet1], [ResNet2], ResNeXts [ResNeXt], DenseNets [DenseNet] and Xception networks [Xception]. Their respective papers provided the number of layers, the number of channels per layer and multiple other details that we never modified in order reduce the number of parameters to tune.

Tuning, Testing and Results

We tested these architectures with different regularization parameters, location of pooling layers and convolution window sizes. The best performance for the different architectures can be found in Table 1. The best performance we obtained during the competition was from a ResNeXt-50 network with a log loss of 0.339. Due to the constraints of the competition, the sub-optimal results of Xception and DenseNet networks may be due to lack of expert tuning time and not an inherent deficiency of these architectures for this problem.

The convolution window size turned out to influence performance dramatically. We found early on that increasing the window size would increase the complexity of the models as well as the score. Our winning ResNeXt-50 network uses window size 64 to obtain its 74.69 score. After the competition we trained the same network with a convolutional window size of 3 and obtained a score of 64.2 which would not have won the challenge.

Merging and Probability Calibration

Merging

As shown in Figure 1, we merged the best Engineering Features (EF) network with the best Deep Learning (DL) network. We merged by taking metrics from both the EF and DL networks as features to go into a new dense neural network. The metrics that worked best were the logit outputs of the last layer of both EF and DL networks as well as the outputs of the penultimate layer of both networks. We believe this to be a novel idea for merging diverse neural networks. We tested using outputs of earlier layers on both networks and didn't obtain a better performance.

The classifier that produced the best results for these new features was a dense neural network. At the input of the merging neural network we use a batch normalization layer [Ioffe] for the features that come from the EF network only. We then concatenate both sets of features and connect them to a dense network that has 2 hidden layers of size 1024 and 512 respectively. The output layer has size 24 which corresponds to the original number of modulations in the challenge.

For reference the code to instantiate the best neural net merging classifier is:

```
from keras.layers import Input,
                         BatchNormalization,
                         Concatenate,
                         Dense,
                         Activation
from keras.models import Model

#Deep Neural Net inputs
main_input1 = Input(shape=(2048,))
main_input2 = Input(shape=(24,))
#Engineering Features Neural Net inputs
auxiliary_input1 = Input(shape=(512,))
auxiliary_input2 = Input(shape=(24,))
#Batch normalizing Engineering Feature layers
x1 = BatchNormalization()(auxiliary_input1)
x2 = BatchNormalization()(auxiliary_input2)
#Concatenate Layers
x = Concatenate([main_input1,main_input2,x1, x2])
#Put through Dense Network
x=Dense(1024, activation='relu', init='he_normal')(x)
x=Dense(512, activation='relu', init='he_normal')(x)
x=Dense(24, init='he_normal')(x)
output=Activation('softmax')(x)
model = Model(inputs=[main_input1,
                      main_input2,
                      auxiliary_input1,
                      auxiliary_input2],
              outputs=output)
```

We tested other types of classifiers that we obtained by using AutoML. The AutoML package we used is TPOT [TPOT1], [TPOT2] which is built on top of scikit-learn. TPOT proposed to use a combination of Linear Support Vector Classification (sklearn.svm.LinearSVC), Naive Bayes for multivariate Bernoulli models (sklearn.naive_bayes.BernoulliNB) and Logistic Regression (sklearn.linear_model.LogisticRegression).

The code to instantiate the best AutoML generated merging classifier is:

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from tpot.builtins import StackingEstimator
import sklearn.feature_selection as sklfs

model = make_pipeline(
 sklfs.VarianceThreshold(threshold=0.1),
 StackingEstimator(
 estimator=BernoulliNB(alpha=100.0)),
```

Classifier(s)	Calibration	Pre-cal score	Post-cal score	Accuracy
Neural Network	Temperature	75.55	75.68	86.94
BernoulliNB and LogisticRegression	isotonic	74.75	74.8	87.2
BernoulliNB and LinearSVC	isotonic	73.9	74.74	87.2
LogisticRegression	isotonic	73.49	74.33	86.93
LinearSVC	isotonic	74.23	74.99	87.22

TABLE 2: *Sub-sampled merging and calibration results.*

Classifier(s)	Calibration	Pre-cal score	Post-cal score	Accuracy
Neural Network	Temperature	75.87	76.42	87.47
BernoulliNB and LogisticRegression	isotonic	74.97	75.14	87.2

TABLE 3: *Complete dataset merging and calibration results.*

```
 LogisticRegression(C=0.01, dual=False, penalty="l1",
                    tol=0.001)
)
```

Probability Calibration

The final step in the pipeline presented in Figure 1 is calibration. Probability calibration consists on modifying the final probabilities without changing the class that corresponds to the highest probability. It uses the 15% cross-validation data to shape the output probabilities to increase the score.

In order to calibrate our merging neural network we used a modification of the temperature scaling approach proposed in [Guo]. The temperature scaling approach finds the optimal temperature scalar to divide the output logits by, that minimizes the log loss on the cross-validation dataset. We extended this method by finding the separate optimal temperature scalars for each predicted modulation type using the cross-validation data. Temperature scaling consistently increased the score of neural nets from 0.3 to 0.6 points.

Calibration of the scikit-learn merging classifiers consisted on using the CalibrateClassifierCV class in scikit-learn [SKCal]. This class implements two different approaches for performing calibration: a parametric approach based on Platt's sigmoid model and a non-parametric approach based on isotonic regression. Our best results were achieved with the isotonic approach which were always between 0.1 to 0.9 points better than the uncalibrated score.

Merging and Calibration Results

The best merging and calibration results are presented in Table 2. These results were obtained by training on the same random subsample of the training dataset of size 144000. Table 3 shows the best merging and calibration results for both neural nets classifiers and scikit-learn classifiers when trained on the full training dataset.

Overall Performance

The accuracy of estimation can be visualized as a confusion matrix, shown in Figures 7 and 8 for the deep learning classifier and the final calibrated and merged classifier respectively.

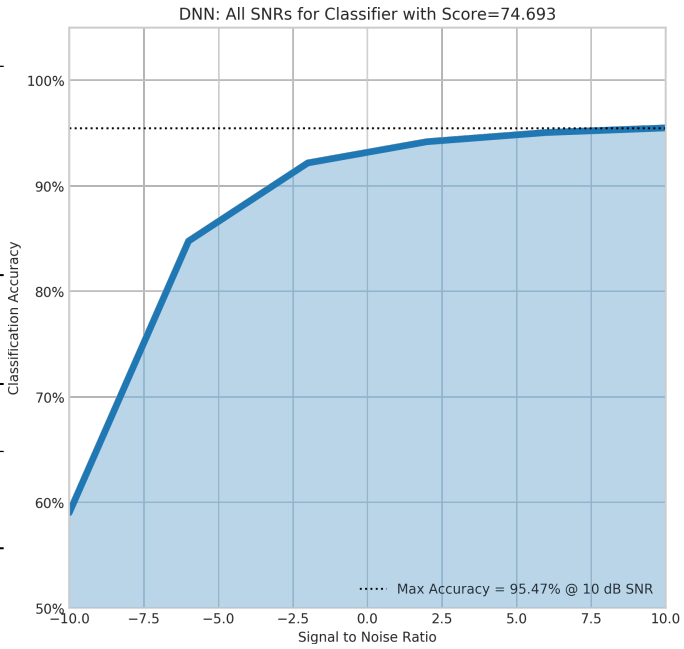


Fig. 3: *Classifier Accuracy vs SNR for deep learning network.*

Each row represents the true waveform, while each column is the estimated probability. The diagonal values correspond to the 'correct' estimate. Brighter colors indicate higher confidence (e.g. the top left square indicates almost 100% correct identification of the BPSK modulation). This view allows us to quickly identify waveforms that are challenging and to see where merging the deep learning classifier with the engineering features classifier helps. Calibration does not improve the confusion matrix since the winning class per sample doesn't change.

The F_1 score (see Challenge Description) provides another view of the same data. Figures 5 and 6 show the performances for the deep learning classifier and the final calibrated and merged classifier respectively. The overall classifier accuracy versus SNR is shown in Figures 3 and 4. Note that we achieve about 63% accuracy even at -10 dB SNR, which is significantly better than previously published results.

Conclusion

This paper showed the variety of ways machine learning techniques in python can be used to dramatically increase the performance of modulation classification algorithms. We presented a performance overview of different deep learning architectures when applied to the one-dimensional RF modulation-classification problem as presented in [Army] and [Mitre]. While the best performing architectures were ResNet and ResNeXt, we would caution against deducing that there is something inherent in those architectures that makes them more suited to the modulation-classification problem. Those algorithms produced the most promising results earlier on and thus, more time was spent running variations of them instead of trying to improve the performance of Xception or DenseNet networks.

This paper also presented a new merging method to fuse different neural networks. The novelty resides in what is being used as the input features of the merging classifiers. We used as inputs not only the results of the final layers of the original networks but the outputs of the last few layers of each of the initial neural networks.

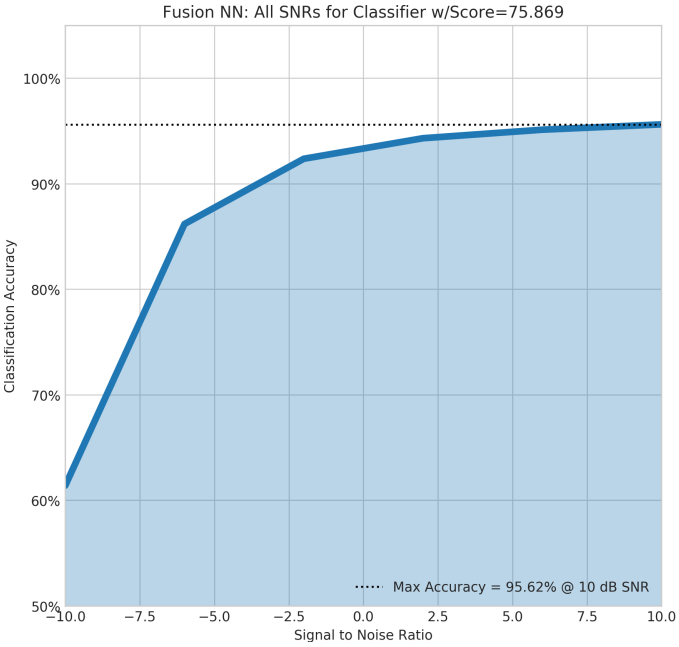


Fig. 4: *Classifier Accuracy vs SNR for final merging network.*

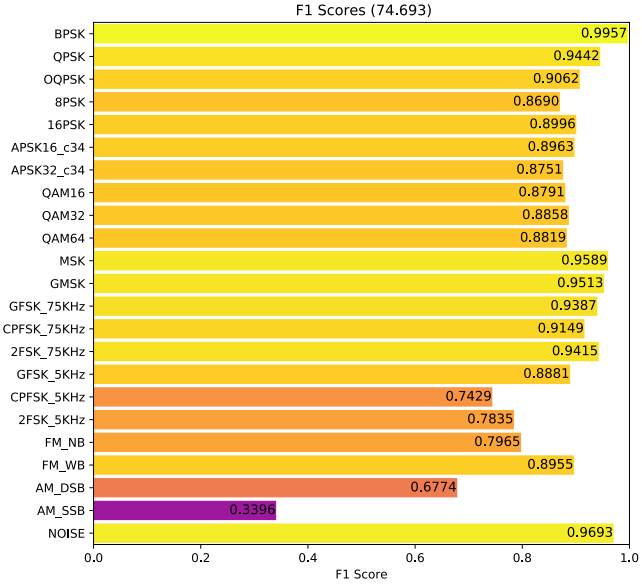


Fig. 5: *F_1 scores for all test data for deep learning network.*

Finally, we showed that calibration techniques can improve the log loss of diverse classifiers. However, it is important to note that the test cases offered by the Challenge are somewhat unrealistic. Real-world scenarios would include non-idealities like those described in [OShea2].

Acknowledgements

The authors would like to thank the Army RCO for creating this interesting challenge as well as our competitors who motivated us to stay up late and reconsider our assumptions.

REFERENCES

[Army] ARMY RCO AI Signal Classification Challenge. (2018). Retrieved from https://www.challenge.gov/challenge/army-signal-classification-challenge/

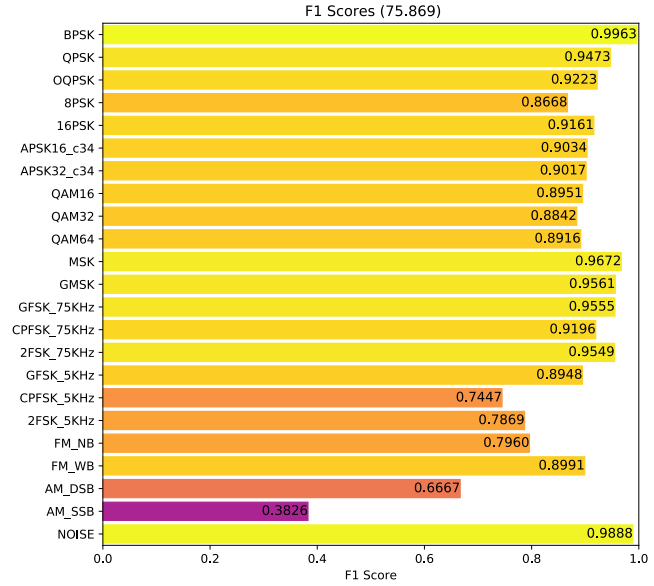


Fig. 6: *F_1 scores for all test data for final merged network.*

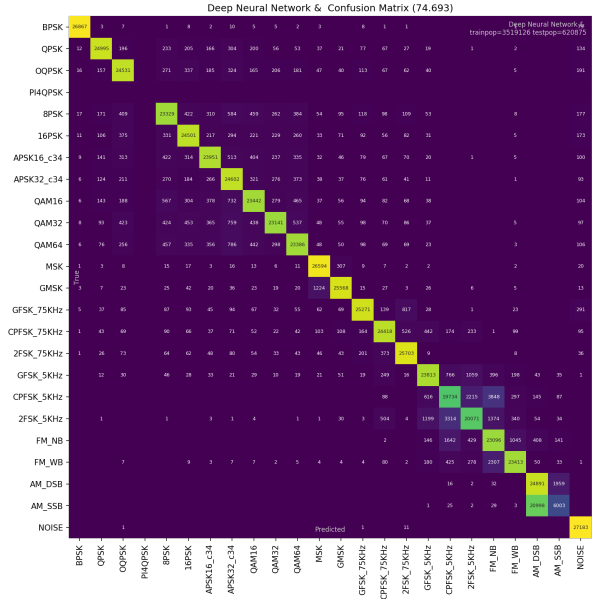


Fig. 7: *Confusion matrix for all test data for deep learning network.*

[Mitre] MITRE Challenge. (2018). Retrieved from https://sites.mitre.org/armychallenge/

[Nandi1] Nandi, Asoke K., and Elsayed Elsayed Azzouz. "Algorithms for automatic modulation recognition of communication signals." IEEE Transactions on communications 46.4 (1998): 431-436. doi:10.1109/26.664294.

[Nandi2] Nandi, A. K., and Elsayed Elsayed Azzouz. "Automatic analogue modulation recognition." Signal processing 46.2 (1995): 211-222. doi:10.1016/0165-1684(95)00083-p.

[Azz1] Azzouz, Elsayed, and Asoke Kumar Nandi. Automatic modulation recognition of communication signals. Springer Science & Business Media, 2013. doi:10.1007/978-1-4757-2469-1.

[Azz2] Azzouz, Elsayed Elsayed, and Asoke Kumar Nandi. "Modulation recognition using artificial neural networks." Automatic Modulation Recognition of Communication Signals. Springer, Boston, MA, 1996. 132-176. doi:10.1007/978-1-4757-2469-1_5.

[OShea1] T. J. O'Shea, J. Corgan, "Convolutional radio modulation recognition networks", CoRR abs/1602.04105, 2016. doi:10.1007/978-3-319-44188-7_16.

[OShea2] T. J. O'Shea, T. Roy and T. C. Clancy. "Over-the-Air Deep Learning Based Radio Signal Classification," in IEEE Journal of

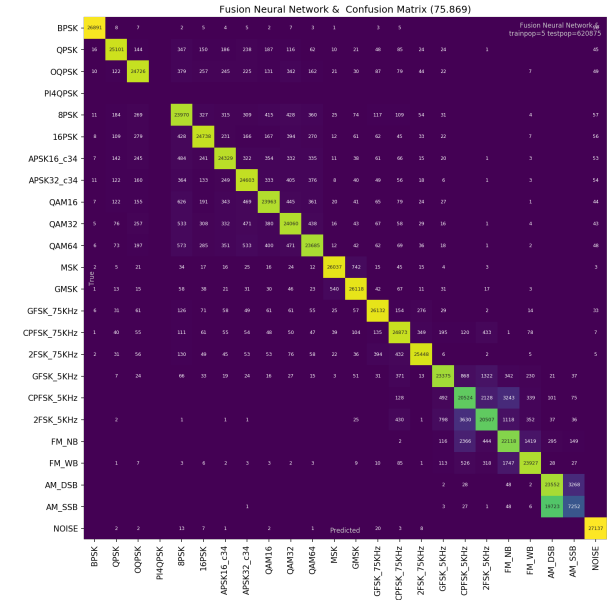


Fig. 8: *Confusion matrix all test data for final merged network.*

Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168-179, Feb. 2018. doi:10.1109/JSTSP.2018.2797022.

[ResNet1] He, K., Zhang, X., Ren, S., Sun, J. "Deep residual learning for image recognition", CVPR arXiv:1512.03385. 2016.

[ResNet2] He, K., Zhang, X., Ren, S., Sun, J. "Identity Mappings in Deep Residual Networks", CVPR arXiv:1603.05027. 2016.

[ResNeXt] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, Kaiming He. "Aggregated Residual Transformations for Deep Neural Networks", CVPR arXiv:1611.05431. 2017.

[Xception] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions", CVPR arXiv:1610.02357. 2016.

[DenseNet] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Deep Residual Learning for Image Recognition", CVPR arXiv:1512.03385. 2015.

[Guo] Chuan Guo, Geoff Pleiss, Yu Sun, Kilian Q. Weinberger. "On Calibration of Modern Neural Networks", ML arXiv:1706.04599. ICML 2017.

[SKCal] Probability calibration. Retreived from https://scikit-learn.org/stable/modules/calibration.html

[TPOT1] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore (2016). Automating biomedical data science through tree-based pipeline optimization. Applications of Evolutionary Computation, pages 123-137. 2016. doi:10.1007/978-3-319-31204-0_9.

[TPOT2] TPOT, a Python Automated Machine Learning tool. Retrieved from https://epistasislab.github.io/tpot/

[Aisbett] Aisbett, Janet. "Automatic modulation recognition using time domain parameters." Signal Processing 13.3 (1987): 323-328. doi:10.1016/0165-1684(87)90130-7.

[Ioffe] Sergey Ioffe, Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". ML arXiv:1502.03167. 2015.

Analyzing Particle Systems for Machine Learning and Data Visualization with freud

Bradley D. Dice[‡*], Vyas Ramasubramani[§], Eric S. Harper[**], Matthew P. Spellings[§], Joshua A. Anderson[§], Sharon C. Glotzer[‡§¶‖]

Abstract—The `freud` Python library analyzes particle data output from molecular dynamics simulations. The library's design and its variety of high-performance methods make it a powerful tool for many modern applications. In particular, `freud` can be used as part of the data generation pipeline for machine learning (ML) algorithms for analyzing particle simulations, and it can be easily integrated with various simulation visualization tools for simultaneous visualization and real-time analysis. Here, we present numerous examples both of using `freud` to analyze nano-scale particle systems by coupling traditional simulational analyses to machine learning libraries and of visualizing per-particle quantities calculated by `freud` analysis methods. We include code and examples of this visualization, showing that in general the introduction of `freud` into existing ML and visualization workflows is smooth and unintrusive. We demonstrate that among Python packages used in the computational molecular sciences, `freud` offers a unique set of analysis methods with efficient computations and seamless coupling into powerful data analysis pipelines.

Index Terms—molecular dynamics, analysis, particle simulation, particle system, computational physics, computational chemistry

Introduction

The availability of "off-the-shelf" molecular dynamics engines (e.g. HOOMD-blue [ALT08], [GNA+15], LAMMPS [Pli95], GROMACS [BvdSvD95]) has made simulating complex systems possible across many scientific fields. Simulations of systems ranging from large biomolecules to colloids are now common, allowing researchers to ask new questions about reconfigurable materials [CDA+18] and develop coarse-graining approaches to access increasing timescales [SZR+19]. Various tools have arisen to facilitate the analysis of these simulations, many of which are immediately interoperable with the most popular simulation tools. The `freud` library is one such analysis package that differentiates itself from others through its focus on colloidal and nano-scale systems.

Due to their diversity and adaptability, colloidal materials are a powerful model system for exploring soft matter physics [GS07]. Such materials are also a viable platform for harnessing photonic [CDA+18], plasmonic [TCLC11], and other useful structurally-derived properties. In colloidal systems, features like particle anisotropy play an important role in creating complex crystal structures, some of which have no atomic analogues [DEG12]. Design spaces encompassing wide ranges of particle morphology [DEG12] and interparticle interactions [AADG18] have been shown to yield phase diagrams filled with complex behavior.

The `freud` Python package offers a unique feature set that targets the analysis of colloidal systems. The library avoids trajectory management and the analysis of chemically bonded structures, which are the province of most other analysis platforms like MDAnalysis and MDTraj (see also 1) [MADWB11], [MBH+15]. In particular, `freud` excels at performing analyses based on characterizing local particle environments, which makes it a powerful tool for tasks such as calculating order parameters to track crystallization or finding prenucleation clusters. Among the unique methods present in `freud` are the potential of mean force and torque, which allows users to understand the effects of particle anisotropy on entropic self-assembly [vAAS+14], [vAKA+14], [KGG16], [HMA+15], [AAM+17], and various tools for identifying and clustering particles by their local crystal environments [TvAG19]. All such tasks are accelerated by `freud`'s extremely fast neighbor finding routines and are automatically parallelized, making it an ideal tool for researchers performing peta- or exascale simulations of particle systems. The `freud` library's scalability is exemplified by its use in computing correlation functions on systems of over a million particles, calculations that were used to illuminate the elusive hexatic phase transition in two-dimensional systems of hard polygons [AAM+17]. More details on the use of `freud` can be found in [RDH+19]. In this paper, we will demonstrate that `freud` is uniquely well-suited to usage in the context of data pipelines for visualization and machine learning applications.

Data Pipelines

The `freud` package is especially useful because it can be organically integrated into a data pipeline. Many research tasks in computational molecular sciences can be expressed in terms of data pipelines; in molecular simulations, such a pipeline typically involves:

1) **Generating** an input file that defines a simulation.
2) **Simulating** the system of interest, saving its trajectory to a file.

* *Corresponding author: bdice@umich.edu*
‡ *Department of Physics, University of Michigan, Ann Arbor*
§ *Department of Chemical Engineering, University of Michigan, Ann Arbor*
** *Department of Materials Science & Engineering, University of Michigan, Ann Arbor*
¶ *Department of Materials Science and Engineering, University of Michigan, Ann Arbor*
‖ *Biointerfaces Institute, University of Michigan, Ann Arbor*

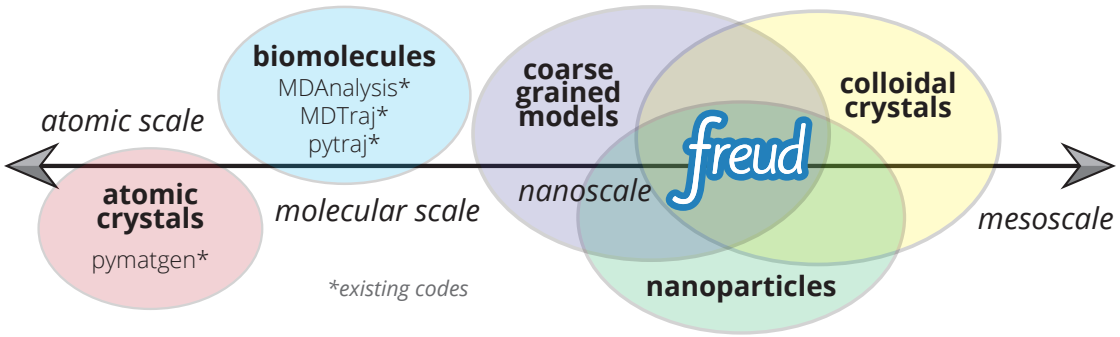


Fig. 1: *Common Python tools for simulation analysis at varying length scales. The freud library is designed for nanoscale systems, such as colloidal crystals and nanoparticle assemblies. In such systems, interactions are described by coarse-grained models where particles' atomic constituents are often irrelevant and particle anisotropy (non-spherical shape) is common, thus requiring a generalized concept of particle "types" and orientation-sensitive analyses. These features contrast the assumptions of most analysis tools designed for biomolecular simulations and materials science.*

3) **Analyzing** the resulting data by computing and storing various quantities.
4) **Visualizing** the trajectory, using colors or styles determined from previous analyses.

However, in modern workflows the lines between these stages is typically blurred, particularly with respect to analysis. While direct visualization of simulation trajectories can provide insights into the behavior of a system, integrating higher-order analyses is often necessary to provide real-time interpretable visualizations in that allow researchers to identify meaningful features like defects and ordered domains of self-assembled structures. Studies of complex systems are also often aided or accelerated by a real-time coupling of simulations with on-the-fly analysis. This simultaneous usage of simulation and analysis is especially relevant because modern machine learning techniques frequently involve wrapping this pipeline entirely within a higher-level optimization problem, since analysis methods can be used to construct objective functions targeting a specific materials design problem, for instance.

Following, we provide demonstrations of how `freud` can be integrated with popular tools in the scientific Python ecosystem like TensorFlow, Scikit-learn, SciPy, or Matplotlib. In the context of machine learning algorithms, we will discuss how the analyses in `freud` can reduce the 6N-dimensional space of particle positions and orientations into a tractable set of features that can be fed into machine learning algorithms. We will further show that `freud` can be used for visualizations even outside of scripting contexts, enabling a wide range of forward-thinking applications including Jupyter notebook integrations, versatile 3D renderings, and integration with various standard tools for visualizing simulation trajectories. These topics are aimed at computational molecular scientists and data scientists alike, with discussions of real-world usage as well as theoretical motivation and conceptual exploration. The full source code of all examples in this paper can be found online[1].

Performance and Integrability

Using `freud` to compute features for machine learning algorithms and visualization is straightforward because it adheres to a UNIX-like philosophy of providing modular, composable features. This design is evidenced by the library's reliance on NumPy arrays [Oli06] for all inputs and outputs, a format that is naturally integrated with most other tools in the scientific Python ecosystem. In general, the analyses in `freud` are designed around analyses of raw particle trajectories, meaning that the inputs are typically $(N,3)$ arrays of particle positions and $(N,4)$ arrays of particle orientations, and analyses that involve many frames over time use *accumulate* methods that are called once for each frame. This general approach enables `freud` to be used for a range of input data, including molecular dynamics and Monte Carlo simulations as well as experimental data (e.g. positions extracted via particle tracking) in both 3D and 2D. The direct usage of numerical arrays indicates a different usage pattern than that of tools, such as MDAnalysis [MADWB11] and MDTraj [MBH+15], for which trajectory parsing is a core feature. Due to the existence of many such tools which are capable of reading simulation engines' output files, as well as certain formats like `gsd`[2] that provide their own parsers, `freud` eschews any form of trajectory management and instead relies on other tools to provide input arrays. If input data is to be read from a file, binary data formats such as `gsd` or NumPy's `npy` or `npz` are strongly preferred for efficient I/O. Though it is possible to use a library like Pandas to load data stored in a comma-separated value (CSV) or other text-based data format, such files are often much slower when reading and writing large numerical arrays. Decoupling `freud` from file parsing and specific trajectory representations allows it to be efficiently integrated into simulations, machine learning applications, and visualization toolkits with no I/O overhead and limited additional code complexity, while the universal usage of NumPy arrays makes such integrations very natural.

In keeping with this focus on composable features, `freud` also abstracts and directly exposes the task of finding particle neighbors, the task most central to all other analyses in `freud`. Since neighbor finding is a common need, the neighbor finding routines in `freud` are highly optimized and natively support periodic systems, a crucial feature for any analysis of particle simulations (which often employ periodic boundary conditions). In figure 2, a comparison is shown between the neighbor finding algorithms in `freud` and SciPy [JOPo01]. For each system size, N particles are uniformly distributed in a 3D periodic cube such that each particle has an average of 12 neighbors within a distance of $r_{cut} = 1.0$. Neighbors are found for each particle by

1. https://github.com/glotzerlab/freud-examples
2. https://github.com/glotzerlab/gsd

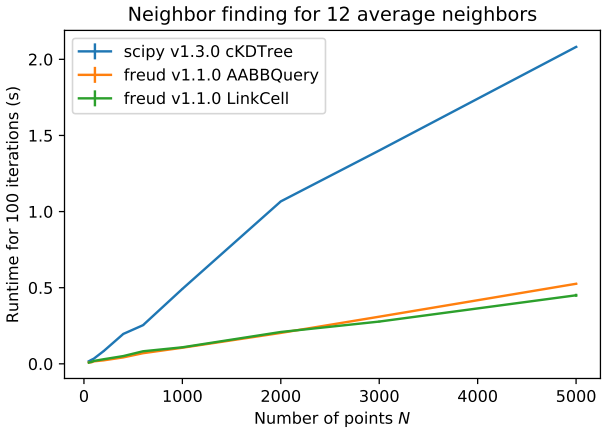


Fig. 2: *Comparison of runtime for neighbor finding algorithms in* `freud` *and SciPy for varied system sizes. See text for details.*

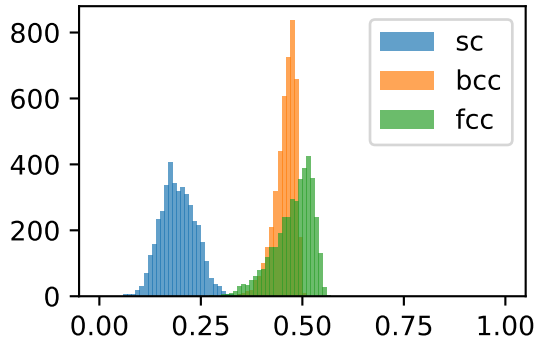


Fig. 3: *Histogram of the Steinhardt Q_6 order parameter for 4000 particles in simple cubic, body-centered cubic, and face-centered cubic structures with added Gaussian noise.*

searching within the cutoff distance r_{cut}. The methods compared are `scipy.spatial.cKDTree`'s `query_ball_tree`, `freud.locality.AABBQuery`'s `queryBall`, and `freud.locality.LinkCell`'s `compute`. The benchmarks were performed with 5 replicates on a 3.6 GHz Intel Core i3-8100B processor with 16 GB 2667 MHz DDR4 RAM.

Evidently, `freud` performs very well on this core task and scales well to larger systems. The parallel C++ backend implemented with Cython and Intel Threading Building Blocks makes `freud` perform quickly even for large systems [BBC+11], [Int18]. Furthermore, `freud` supports periodicity in arbitrary triclinic volumes, a common feature found in many simulations. This support distinguishes it from other tools like `scipy.spatial.cKDTree`, which only supports cubic boxes. The fast neighbor finding in `freud` and the ease of integrating its outputs into other analyses not only make it easy to add fast new analysis methods into `freud`, they are also central to why `freud` can be easily integrated into workflows for machine learning and visualization.

Machine Learning

A wide range of problems in soft matter and nano-scale simulations have been addressed using machine learning techniques, such as crystal structure identification [SG18]. In machine learning workflows, `freud` is used to generate features, which are then used in classification or regression models, clusterings, or dimensionality reduction methods. For example, Harper et al. used `freud` to compute the cubatic order parameter and generate high-dimensional descriptors of structural motifs, which were visualized with t-SNE dimensionality reduction [HWG19], [vdMH08]. The library has also been used in the optimization and inverse design of pair potentials [AADG18], to compute fitness functions based on the radial distribution function. The open-source `pythia`[3] library offers a number of descriptor sets useful for crystal structure identification, leveraging `freud` for fast computations. Included among the descriptors in `pythia` are quantities based on bond angles and distances, spherical harmonics, and Voronoi diagrams.

Computing a set of descriptors tuned for a particular system of interest (e.g. using values of Q_l, the higher-order Steinhardt W_l parameters, or other order parameters provided by `freud`) is possible with just a few lines of code. Descriptors like these (exemplified in the `pythia` library) have been used with TensorFlow for supervised and unsupervised learning of crystal structures in complex phase diagrams [SG18], [AAB+15].

Another useful module for machine learning with `freud` is `freud.cluster`, which uses a distance-based cutoff to locate clusters of particles while accounting for 2D or 3D periodicity. Locating clusters in this way can identify crystalline grains, helpful for building a training set for machine learning models.

To demonstrate a concrete example, we focus on a common challenge in molecular sciences: identifying crystal structures. Recently, several approaches have been developed that use machine learning for detecting ordered phases [SCKL15], [SG18], [FSM19], [SNR83], [LD08]. The Steinhardt order parameters are often used as a structural fingerprint, and are derived from rotationally invariant combinations of spherical harmonics. In the example below, we create face-centered cubic (fcc), body-centered cubic (bcc), and simple cubic (sc) crystals with added Gaussian noise, and use Steinhardt order parameters with a support vector machine to train a simple crystal structure identifier. Steinhardt order parameters characterize the spherical arrangement of neighbors around a central particle, and combining values of Q_l for a range of l often gives a unique signature for simple crystal structures. This example demonstrates a simple case of how `freud` can be used to help solve the problem of structural identification, which often requires a sophisticated approach for complex crystals.

In figure 3, we show the distribution of Q_6 values for sample structures with 4000 particles. Here, we demonstrate how to compute the Steinhardt Q_6, using neighbors found via a periodic Voronoi diagram. Neighbors with small facets in the Voronoi polytope are filtered out to reduce noise.

```
import freud
import numpy as np
from util import make_fcc

def get_features(box, positions, structure):
    # Create a Voronoi compute object
    voro = freud.voronoi.Voronoi(
        box, buff=max(box.L)/2)
    voro.computeNeighbors(positions)

    # Filter the Voronoi NeighborList
    nlist = voro.nlist
    nlist.filter(nlist.weights > 0.1)
```

3. https://github.com/glotzerlab/pythia

```
    # Compute Steinhardt order parameters
    features = {}
    for l in [4, 6, 8, 10, 12]:
        ql = freud.order.LocalQl(
             box, rmax=max(box.L)/2, l=l)
        ql.compute(positions, nlist)
        features['q{}'.format(l)] = ql.Ql.copy()

    return features

# Create a freud box object and an array of
# 3D positions for a face-centered cubic
# structure with 4000 particles
fcc_box, fcc_positions = make_fcc(
    nx=10, ny=10, nz=10, noise=0.1)

structures = {}
structures['fcc'] = get_features(
    fcc_box, fcc_positions, 'fcc')
# ... repeat for all structures
```

Then, using Pandas and Scikit-learn, we can train a support vector machine to identify these structures:

```
# Build dictionary of DataFrames,
# labeled by structure
structure_dfs = {}
for i, struct in enumerate(structures):
    df = pd.DataFrame.from_dict(structures[struct])
    df['class'] = i
    structure_dfs[struct] = df

# Combine DataFrames for input to SVM
df = pd.concat(structure_dfs.values())
df = df.reset_index(drop=True)

from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# We use the normalized Steinhardt order parameters
# to predict the crystal structure
X = df.drop('class', axis=1).values
X = normalize(X)
y = df['class'].values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42)

svm = SVC()
svm.fit(X_train, y_train)
print('Score:', svm.score(X_test, y_test))
# The model is ~98% accurate.
```

To interpret crystal identification models like this, it can be helpful to use a dimensionality reduction tool such as Uniform Manifold Approximation and Projection (UMAP) [MH18], as shown in figure 4. The low-dimensional UMAP projection shown is generated directly from the Pandas `DataFrame`:

```
from umap import UMAP
umap = UMAP()

# Project the high-dimensional descriptors
# to a two dimensional manifold
data = umap.fit_transform(df)
plt.plot(data[:, 0], data[:, 1])
```

Visualization

Many analyses performed by the `freud` library provide a `plot(ax=None)` method (new in v1.2.0) that allows their computed quantities to be visualized with Matplotlib. Additionally, these plottable analyses offer IPython representations, allowing Jupyter notebooks to render a graph such as a radial distribution function $g(r)$ just by returning the compute object at the end of a cell. Analyses like the radial distribution function or correlation functions return data that is binned as a one-dimensional histogram -- these are visualized with a line graph via `matplotlib.pyplot.plot`, with the bin locations and bin counts given by properties of the compute object. Other classes provide multi-dimensional histograms, like the Gaussian density or Potential of Mean Force and Torque, which are plotted with `matplotlib.pyplot.imshow`.

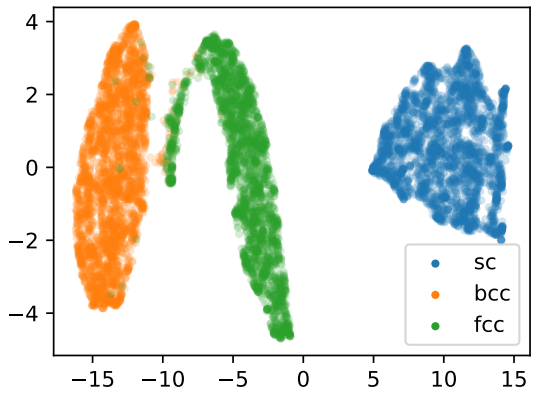


Fig. 4: *UMAP of particle descriptors computed for simple cubic, body-centered cubic, and face-centered cubic structures of 4000 particles with added Gaussian noise. The particle descriptors include Q_l for $l \in \{4, 6, 8, 10, 12\}$. Some noisy configurations of bcc can be confused as fcc and vice versa, which accounts for the small number of errors in the support vector machine's test classification.*

The most complex case for visualization is that of per-particle properties, which also comprises some of the most useful features in `freud`. Quantities that are computed on a per-particle level can be continuous (e.g. Steinhardt order parameters) or discrete (e.g. clustering, where the integer value corresponds to a unique cluster ID). Continuous quantities can be plotted as a histogram over particles, but typically the most helpful visualizations use these quantities with a color map assigned to particles in a two- or three-dimensional view of the system itself. For such particle visualizations, several open-source tools exist that interoperate well with `freud`. Below are examples of how one can integrate `freud` with `plato`[4], `fresnel`[5], and OVITO[6] [Stu10].

plato

`plato` is an open-source graphics package that expresses a common interface for defining two- or three-dimensional scenes which can be rendered as an interactive Jupyter widget or saved to a high-resolution image using one of several backends (PyThreejs, Matplotlib, `fresnel`, POVray[7], and Blender[8], among others). Below is an example of how to render particles from a HOOMD-blue snapshot, colored by the density of their local environment [ALT08], [GNA+15]. The result is shown in figure 5.

```
import plato
import plato.draw.pythreejs as draw
import numpy as np
```

4. https://github.com/glotzerlab/plato
5. https://github.com/glotzerlab/fresnel
6. https://ovito.org/
7. https://www.povray.org/
8. https://www.blender.org/

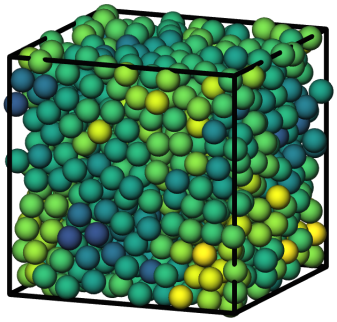


Fig. 5: *Interactive visualization of a Lennard-Jones particle system, rendered in a Jupyter notebook using* `plato` *with the* `pythreejs` *backend.*

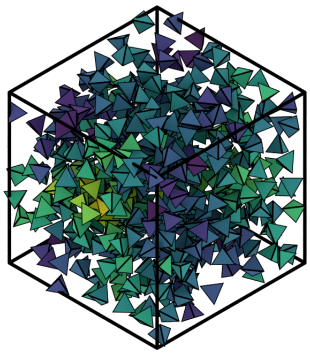


Fig. 6: *Hard tetrahedra colored by local density, path traced with* `fresnel`*.*

```
import matplotlib.cm
import freud
from sklearn.preprocessing import minmax_scale

# snap comes from a previous HOOMD-blue simulation
box = freud.box.Box.from_box(snap.box)
positions = snap.particles.position

# Compute the local density of each particle
ld = freud.density.LocalDensity(
    r_cut=3.0, volume=1.0, diameter=1.0)
ld.compute(box, positions)

# Create a scene for visualization,
# colored by local density
radii = 0.5 * np.ones(len(positions))
colors = matplotlib.cm.viridis(
    minmax_scale(ld.density))
spheres_primitive = draw.Spheres(
    positions=positions,
    radii=radii,
    colors=colors)
scene = draw.Scene(spheres_primitive, zoom=2)
scene.show()  # Interactive view in Jupyter
```

fresnel

`fresnel`[9] is a GPU-accelerated ray tracer designed for particle simulations, with customizable material types and scene lighting, as well as support for a set of common anisotropic shapes. Its feature set is especially well suited for publication-quality graphics. Its use of ray tracing also means that an image's rendering time scales most strongly with the image size, instead of the number of particles -- a desirable feature for extremely large simulations. An example of how to integrate `fresnel` is shown below and rendered in figure 6.

```
# Generate a snapshot of tetrahedra using HOOMD-blue
import hoomd
import hoomd.hpmc
hoomd.context.initialize('')

# Create an 8x8x8 simple cubic lattice
system = hoomd.init.create_lattice(
    unitcell=hoomd.lattice.sc(a=1.5), n=8)

# Create tetrahedra, configure HPMC integrator
mc = hoomd.hpmc.integrate.convex_polyhedron(seed=123)
mc.set_params(d=0.2, a=0.1)
vertices = [( 0.5, 0.5, 0.5),
            (-0.5,-0.5, 0.5),
            (-0.5, 0.5,-0.5),
            ( 0.5,-0.5,-0.5)]
mc.shape_param.set('A', vertices=vertices)

# Run for 5,000 steps
hoomd.run(5e3)
snap = system.take_snapshot()

# Import analysis & visualization libraries
import fresnel
import freud
import matplotlib.cm
from matplotlib.colors import Normalize
import numpy as np
device = fresnel.Device()

# Compute local density and prepare geometry
poly_info = \
    fresnel.util.convex_polyhedron_from_vertices(
        vertices)
positions = snap.particles.position
orientations = snap.particles.orientation
box = freud.box.Box.from_box(snap.box)
ld = freud.density.LocalDensity(3.0, 1.0, 1.0)
ld.compute(box, positions)
colors = matplotlib.cm.viridis(
    Normalize()(ld.density))
box_points = np.asarray([
    box.makeCoordinates(
        [[0, 0, 0], [0, 0, 0], [0, 0, 0],
         [1, 1, 0], [1, 1, 0], [1, 1, 0],
         [0, 1, 1], [0, 1, 1], [0, 1, 1],
         [1, 0, 1], [1, 0, 1], [1, 0, 1]]),
    box.makeCoordinates(
        [[1, 0, 0], [0, 1, 0], [0, 0, 1],
         [1, 0, 0], [0, 1, 0], [1, 1, 1],
         [1, 1, 1], [0, 1, 0], [0, 0, 1],
         [0, 0, 1], [1, 1, 1], [1, 0, 0]])])

# Create scene
scene = fresnel.Scene(device)
geometry = fresnel.geometry.ConvexPolyhedron(
    scene, poly_info,
    position=positions,
    orientation=orientations,
    color=fresnel.color.linear(colors))
geometry.material = fresnel.material.Material(
    color=fresnel.color.linear([0.25, 0.5, 0.9]),
    roughness=0.8, primitive_color_mix=1.0)
geometry.outline_width = 0.05
box_geometry = fresnel.geometry.Cylinder(
    scene, points=box_points.swapaxes(0, 1))
box_geometry.radius[:] = 0.1
box_geometry.color[:] = np.tile(
    [0, 0, 0], (12, 2, 1))
box_geometry.material.primitive_color_mix = 1.0
scene.camera = fresnel.camera.fit(
    scene, view='isometric', margin=0.1)
scene.lights = fresnel.light.lightbox()
```

9. https://github.com/glotzerlab/fresnel

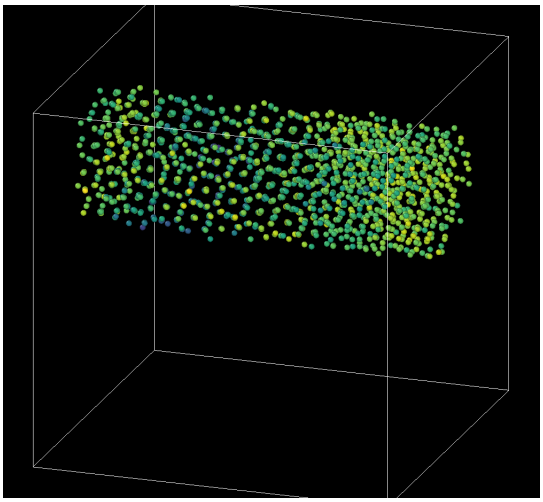


Fig. 7: *A crystalline grain identified using* `freud`*'s* `LocalDensity` *module and cut out for display using OVITO. The image shows a tP30-CrFe structure formed from an isotropic pair potential optimized to generate this structure [AADG18].*

```
# Path trace the scene
fresnel.pathtrace(scene, light_samples=64,
                  w=800, h=800)
```

OVITO

OVITO is a GUI application with features for particle selection, making movies, and support for many trajectory formats [Stu10]. OVITO has several built-in analysis functions (e.g. Polyhedral Template Matching), which complement the methods in `freud`. The Python scripting functionality built into OVITO enables the use of `freud` modules, demonstrated in the code below and shown in figure 7.

```
import freud

def modify(frame, input, output):

    if input.particles != None:
        box = freud.box.Box.from_matrix(
            input.cell.matrix)
        ld = freud.density.LocalDensity(
            r_cut=3, volume=1, diameter=0.05)
        ld.compute(box, input.particles.position)
        output.create_user_particle_property(
            name='LocalDensity',
            data_type=float,
            data=ld.density.copy())
```

Conclusions

The `freud` library offers a unique set of high-performance algorithms designed to accelerate the study of nanoscale and colloidal systems. These algorithms are enabled by a fast, easy-to-use set of tools for identifying particle neighbors, a common first step in nearly all such analyses. The efficiency of both the core neighbor finding algorithms and the higher-level analyses makes them suitable for incorporation into real-time visualization environments, and, in conjunction with the transparent NumPy-based interface, allows integration into machine learning workflows using iterative optimization routines that require frequent recomputation of these analyses. The use of `freud` for real-time visualization has the potential to simplify and accelerate existing simulation visualization pipelines, which typically involve slower and less easily integrable solutions to performing real-time analysis during visualization. The application of `freud` to machine learning, on the other hand, opens up entirely new avenues of research based on treating well-known analyses of particle simulations as descriptors or optimization targets. In these ways, `freud` can facilitate research in the field of computational molecular science, and we hope these examples will spark new ideas for scientific exploration in this field.

Getting `freud`

The `freud` library is tested for Python 2.7 and 3.5+ and is compatible with Linux, macOS, and Windows. To install `freud`, execute

```
conda install -c conda-forge freud
```

or

```
pip install freud-analysis
```

Its source code is available on GitHub[10] and its documentation is available via ReadTheDocs[11].

Acknowledgments

Thanks to Jin Soo Ihm for benchmarking the neighbor finding features of `freud` against SciPy. The `freud` library's code development and public code releases are supported by the National Science Foundation, Division of Materials Research under a Computational and Data-Enabled Science & Engineering Award # DMR 1409620 (2014-2018) and the Office of Advanced Cyberinfrastructure Award # OAC 1835612 (2018-2021). B.D. is supported by a National Science Foundation Graduate Research Fellowship Grant DGE 1256260. M.P.S acknowledges funding from the Toyota Research Institute; this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity. Data for Figure 7 generated on the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575; XSEDE award DMR 140129.

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/.

[AADG18] Carl S. Adorf, James Antonaglia, Julia Dshemuchadse, and Sharon C. Glotzer. Inverse design of simple pair potentials for the self-assembly of complex structures. *The Journal of Chemical Physics*, 149(20):204102, 11 2018. doi:10.1063/1.5063802.

[AAM+17] Joshua A. Anderson, James Antonaglia, Jaime A. Millan, Michael Engel, and Sharon C. Glotzer. Shape and Symmetry Determine Two-Dimensional Melting Transitions of Hard Regular Polygons. *Physical Review X*, 7(2):021001, 4 2017. doi:10.1103/PhysRevX.7.021001.

10. https://github.com/glotzerlab/freud
11. https://freud.readthedocs.io/

[ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. doi:https://doi.org/10.1016/j.jcp.2008.01.047.

[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, 3 2011. doi:10.1109/MCSE.2010.118.

[BvdSvD95] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3):43–56, 9 1995. doi:10.1016/0010-4655(95)00042-E.

[CDA+18] Rose K. Cersonsky, Julia Dshemuchadse, James A. Antonaglia, Greg van Anders, and Sharon C. Glotzer. Pressure-Tunable Photonic Band Gaps in an Entropic Colloidal Crystal. *Physical Review Materials*, 2:125201, 2018. doi:10.1103/PhysRevMaterials.2.125201.

[DEG12] Pablo F. Damasceno, Michael Engel, and Sharon C. Glotzer. Predictive Self-Assembly of Polyhedra into Complex Structures. *Science*, 337(6093):453–457, 7 2012. doi:10.1126/science.1220869.

[FSM19] Maxwell Fulford, Matteo Salvalaglio, and Carla Molteni. DeepIce: a Deep Neural Network Approach to Identify Ice and Water Molecules. *Journal of Chemical Information and Modeling*, page acs.jcim.9b00005, 3 2019. doi:10.1021/acs.jcim.9b00005.

[GNA+15] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications*, 192:97–107, 2015. doi:10.1016/j.cpc.2015.02.028.

[GS07] Sharon C. Glotzer and Michael J. Solomon. Anisotropy of building blocks and their assembly into complex structures. *Nature Materials*, 6:557–562, Aug 2007. URL: https://doi.org/10.1038/nmat1949.

[HMA+15] Eric S. Harper, Ryan L. Marson, Joshua A. Anderson, Greg van Anders, and Sharon C. Glotzer. Shape allophiles improve entropic assembly. *Soft Matter*, 11(37):7250–7256, 9 2015. doi:10.1039/C5SM01351H.

[HWG19] Eric S. Harper, Brendon Waters, and Sharon C. Glotzer. Hierarchical self-assembly of hard cube derivatives. *Soft Matter*, 15:3733–3739, 2019. doi:10.1039/C8SM02619J.

[Int18] Intel. Intel Threading Building Blocks, 2018. URL: https://www.threadingbuildingblocks.org/.

[JOPo01] Eric Jones, Travis Oliphant, Pearu Peterson, and others. SciPy: Open source scientific tools for Python, 2001. URL: https://www.scipy.org/.

[KGG16] Andrew S. Karas, Jens Glaser, and Sharon C. Glotzer. Using depletion to control colloidal crystal assemblies of hard cuboctahedra. *Soft Matter*, 12(23):5199–5204, 6 2016. doi:10.1039/C6SM00620E.

[LD08] Wolfgang Lechner and Christoph Dellago. Accurate determination of crystal structures based on averaged local bond order parameters. *Journal of Chemical Physics*, 129(11), 2008. doi:10.1063/1.2977970.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, 7 2011. doi:10.1002/jcc.21787.

[MBH+15] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. *Biophysical Journal*, 109(8):1528–1532, 10 2015. doi:10.1016/J.BPJ.2015.08.015.

[MH18] Leland McInnes and John Healy. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. Feb 2018. arXiv:1802.03426.

[Oli06] Travis E. Oliphant. *A guide to NumPy*. Trelgol Publishing, 2006.

[Pli95] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, Mar 1995. doi:10.1006/JCPH.1995.1039.

[RDH+19] Vyas Ramasubramani, Bradley D. Dice, Eric S. Harper, Matthew P Spellings, Joshua A. Anderson, and Sharon C. Glotzer. freud: A Software Suite for High Throughput Analysis of Particle Simulation Data. June 2019. arXiv:1906.06317.

[SCKL15] S S Schoenholz, E D Cubuk, E Kaxiras, and a J Liu. A structural approach to relaxation in glassy liquids. *Nature Physics*, (February):1–11, 2015. doi:10.1038/nphys3644.

[SG18] Matthew Spellings and Sharon C. Glotzer. Machine learning for crystal identification and discovery. *AIChE Journal*, 64(6):2198–2206, 6 2018. doi:10.1002/aic.16157.

[SNR83] Paul J. Steinhardt, David R Nelson, and Marco Ronchetti. Bond-orientational order in liquids and glasses. *Physical Review B*, 28(2), 1983.

[Stu10] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. *Modelling and Simulation in Materials Science and Engineering*, 18(1):015012, 1 2010. doi:10.1088/0965-0393/18/1/015012.

[SZR+19] Anna J Simon, Yi Zhou, Vyas Ramasubramani, Jens Glaser, Arti Pothukuchy, Jimmy Gollihar, Jillian C. Gerberich, Janelle Leggere, Barrett R Morrow, Cheulhee Jung, Sharon C Glotzer, David W Taylor, and Andrew D Ellington. Supercharging enables organized assembly of synthetic biomolecules. *Nature Chemistry*, 11:204–212, 2019. doi:10.1038/s41557-018-0196-3.

[TCLC11] Shawn J. Tan, Michael J. Campolongo, Dan Luo, and Wenlong Cheng. Building plasmonic nanostructures with DNA. *Nature Nanotechnology*, 6(5):268–276, 5 2011. doi:10.1038/nnano.2011.49.

[TvAG19] Erin G. Teich, Greg van Anders, and Sharon C. Glotzer. Identity crisis in alchemical space drives the entropic colloidal glass transition. *Nature Communications*, 10(1):64, 12 2019. doi:10.1038/s41467-018-07977-2.

[vAAS+14] Greg van Anders, N. Khalid Ahmed, Ross Smith, Michael Engel, and Sharon C. Glotzer. Entropically patchy particles: Engineering valence through shape entropy. *ACS Nano*, 8(1):931–940, 2014. doi:10.1021/nn4057353.

[vAKA+14] Greg van Anders, Daphne Klotsa, N. Khalid Ahmed, Michael Engel, and Sharon C. Glotzer. Understanding shape entropy through local dense packing. *Proceedings of the National Academy of Sciences*, 111(45):E4812–E4821, 2014. doi:10.1073/pnas.1418159111.

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

CAF Implementation on FPGA Using Python Tools

Chiranth Siddappa[‡*], Mark Wickert[‡]

Abstract—The purpose of this project is to provide a real time geolocation solution by generating code for the complex ambiguity function (CAF) in a hardware description language (HDL) and the implementation on FPGA hardware. The CAF has many practical applications, the more traditional being radar or sonar type systems. By using scientific Python tools, this project provides a solution for testing signals and the ability to customize modules to target multiple devices. The processing for this implementation will be done on a PYNQ board designed by Xilinx. The PYNQ board provides a Zynq chip which has both an ARM CPU and FPGA fabric. All required mathematical operations for the CAF are returned to the user through Python classes which produce synthesizable code in the Verilog HDL. The Python classes use Jinja templates integrated into the Verilog code to allow for configuration changes that a user will need to change for investigation and simulation, development, and test. Helper methods are included in the package to help simulation of the HDL such as quantization, complex data reading and writing, and methods to verify the data using quantized values.

Index Terms—complex, ambiguity, function, overlay, verilog, jinja, jupyter, xilinx, fpga, zynq, pynq, linux

Introduction

In this investigation, the pre-processing steps of downsampling and filtering are simulated and considered outside of the scope of this project. In the case of geolocation systems, the use of collectors and reference emitters are used to create geometries that will allow for the detection of Doppler and movement in the signal. The Doppler is used to calculate a frequency difference of arrival (FDOA). Then, cross correlations can be used to determine the time delay by denoting the peak location of the resulting output as a time delay of arrival (TDOA). The goal of this project is to be able to provide a real time solution for FDOA and TDOA. The basic algorithm for calculating the complex ambiguity function for time difference of arrival and frequency offset (CAF) has been well known since the early 1980's [Ste81]. In many radio frequency applications, there is a need to find a time lag of the signal or the frequency offset of a signal. The reader would be familiar with a form of frequency offset known as Doppler as a common example. The CAF is the joint time offset and frequency offset generalization. The CAF was mainly used first for radar and sonar type processing for locating objects using a method known as active echo location [KPK81]. In this scenario, a matched filter design would be used to ensure that the signals match [Wei94]. More commonly with newer radio frequency systems such as GPS, similar but orthogonal signals are transmitted in the same frequency range. Because of the property of orthogonal signals not cross correlating they do not collide with one another, and they are an optimal signal type for testing this application [ZT08].

* Corresponding author: csiddapp@uccs.edu
‡ University of Colorado Colorado Springs

Motivation

The CAF has many practical applications, the more traditional being the aforementioned radar and sonar type systems, with a similar use case in image processing. The use of cross-correlations in the form of the dot product to find similarities is the same theoretical basis for our use in geolocation. In the particular case of geolocation systems, the use of collectors and reference emitters are used to create geometries that will allow for the detection of Doppler and movement in the signal. This method of calculation has yet to be simplified. Currently GPU's have been employed as the main workhorse due to the availability as a co-prorcessor. But the use of the FPGA has always been an attractive alternative due to the high configurability of the hardware options, but comes with much higher up front design cost [HP17]. For design cost, we are primarily concerned with the development time for code that can be written in C syntax in the form of OpenCL or CUDA for a GPU, as compared to using an HDL which will require background in digital logic and testing that must occur on hardware directly.

To geolocate a signal emitter's location the Doppler is used to calculate a frequency difference of arrival (FDOA) which represents a satellite's drift. Then, cross correlations can be used to determine the time delay by denoting the peak of the resulting output as a time delay of arrival (TDOA). The refernce signal will be different for every use case, which motivates the need to ensure that the resulting Verilog hardware description language (HDL) module output can also be produced to match necessary configurations [ver01]. This became a project goal motivated off work done by other projects to be able to produce code in other languages [Sym]. Thus, the solution provided must be able to be reconfigured based off of different needs. The processing for this system will be targeted to a PYNQ board manufactured by Xilinx, but has been designed such that it can be synthesized to any target device. All Verilog HDL modules that are produced by the Python classes conform to the AXI bus standards of interfacing [Arm17]. This allows for a streamlined plug and play connection between all the modules and is the basis of the templating that is implemented with the help of Jinja.

Starting Point

The main concepts necessary for the understanding of the CAF are topics that are covered in Modern Digital Signal Processing, Communication Systems, and a digital design course. These concepts

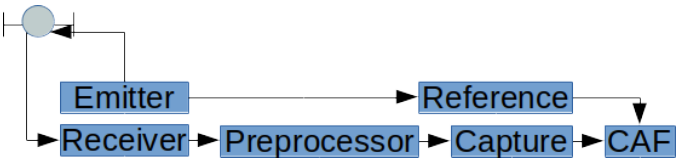


Fig. 1: *Satellite Block Diagram for Emitter and Receiver.*

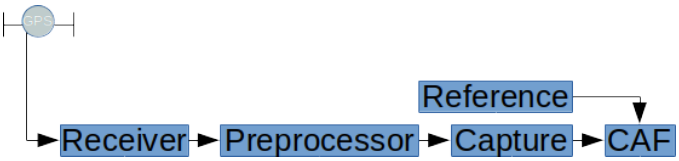


Fig. 2: *Satellite Block Diagram for CAF with GPS Signal.*

would be the Fast Fourier Transform (FFT), integration in both infinite and discrete forms, frequency shifting, and digital design. This project will show a working implementation of digital design HDL modules implementing the logic accurately with this given knowledge. Given the mathematical background of this project, it is crucial to have a way to test implementations against theory. This is the motivation for the discussion of using Python to help generate code and test benches.

Project Overview

The goal of this project was to implement the CAF in an HDL such that the end product can be targeted to any device. The execution of this goal was taken as a bottom up design approach, and as such the discussion starts from small elements to larger ones. The steps taken were in the following order:

1) Obtain and generate a working CAF simulation
2) Break simulation into workable modules
3) Design modules
4) Verify and generate with test benches
5) Assemble larger modules
6) Synthesize and Implement using Vivado for the PYNQ-Z1 board

Complex Ambiguity Function

An example of the signal path in the satellite receiver scenario is described by Fig. 1. In this case, an emitted signal is sent to a satellite, and then received and captured by an RF receiver. Some amount of offset is expected to have happened during the physical relay of the signal back to a receiver within the broadcast area of the satellite. The signal is then downconverted and filtered, and then sent to the CAF via a capture buffer. While a signal is sent through an upconverter and relayed to the satellite, a copy of the same signal must be stored away as a reference to compute the TDOA and FDOA. Both the reference and capture blocks are abstractions, and have individual modules written in Verilog to handle the storage of these signals.

Another very specific example of the satellite receiver scenario is described by Fig. 2. In this scenario, we see that no emitter exists, yet a reference signal is able to be sent to the CAF for TDOA and FDOA calculations. This is because GPS signals use a PRN sequence as ranging codes, and the taps for the signals are provided to the user [Na18]. This provides a significant processing gain as the expected sequence can be computed in real time or stored locally. This project takes advantage of these signals through the use of gps-helper [WSa].

As a basis for what the rest of this paper is describing, an overview of the CAF and the various forms of computing are provided.

The general form of the CAF is:

$$\chi(\tau,f) = \int_{-\infty}^{\infty} s(t)s^*(t-\tau)e^{i2\pi(f/f_s)t}\,dt, \quad \frac{-f_s}{2} < f < \frac{f_s}{2}$$

The equation describes both a time offset τ and a frequency offset f that are used to create a surface. The frequency shift f is bounded by half the sampling rate. The discrete form is a little simpler, and lends itself to the direct implementation [Har05]:

$$\chi(k,f) = \sum_{n=0}^{N-1} s[n]s^*[n-k]e^{i2\pi(f/f_s)(n/N)}, \quad \frac{-f_s}{2} < f < \frac{f_s}{2}$$

where N is the signal capture window length, f_s is the sampling rate in Hz making f have units of Hz and kD is a discrete time offset in samples with sample period $1/f_s$. In both the continuous and discrete-time domains, χ is a function of both time offset and frequency offset. The symbol s represents the signal in question, generally considered to be the reference signal. The accompanying s^* is the complex conjugate and time shifted signal.

As an example, a signal that was not time shifted would simply be the autocorrelation [ZT08]. It is referred to as the received signal in this context, and it is the signal that is used to determine both the time and frequency offset. To determine this offset, we are attempting to shift the signal as close as possible to the original reference signal. The time offset is what allows for the computation of a TDOA, and the frequency offset is what allows for the computation of the FDOA.

In this implementation, the frequency offset is created by a signal generator and a complex multiply module that are both configurable. Once this offset has been applied, a cross-correlation is applied directly in the form of the dot product. This eliminates the costly implementation case where an FFT and an inverse FFT are used to produce a result. The signal generator can supply a specified frequency step and accuracy with configuration of the signal generator class [Sida]. An example of the signal generator is shown in Fig. 9. The resulting spectrum is shown in Fig. 8. This satisfies the frequency (f) portion of the equation. The complex multiply module is similarly configurable for different bit widths through the complex multiply generator class [Sida]. An example CAF surface is provided in Fig. 3 showing how the energy of the signal is spread over both frequency and time. This type of visualization is very useful for real-world signals with associated noise. In this project, care was taken in truncation choices to ensure that the correlation summation ensures signal energy retention. In this project, the CAF module that has been implemented will return a time offset index and frequency offset index back to the user based off provided build parameters shown in the code listing for the Python class *CAF*, described in a later section for the CAF Module. When writing the module, all simulation and testing was done at the sample by sample level to ensure validity so the CAF surface was not used in testing. A method for computing the CAF using the dot product and frequency shifts has been published to the package. This implementation is specific to this project in that it uses a sample size that is twice that of the reference signal for the computation. A sample output slice will be shown in the Experiments section for the CAF module in Fig. 16.

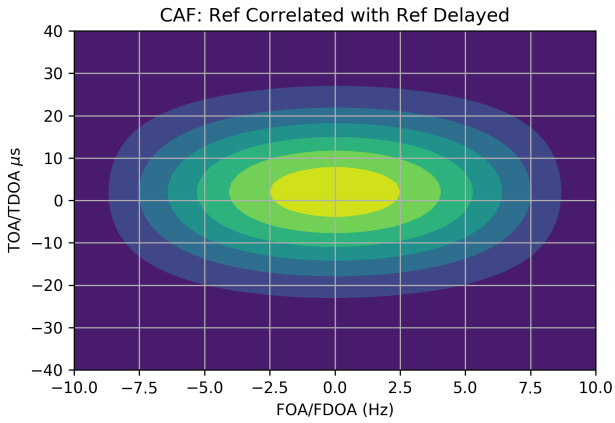


Fig. 3: *CAF Surface Example.*

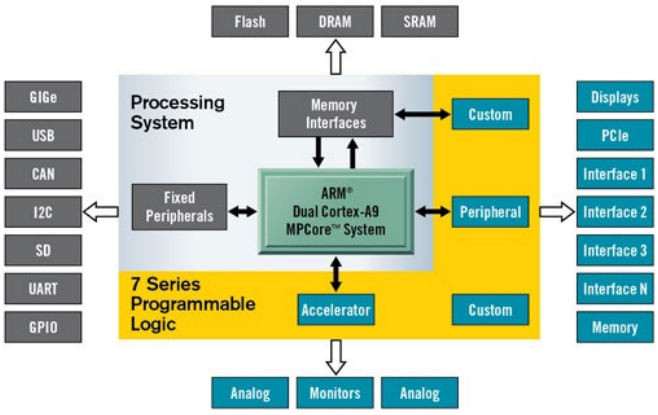


Fig. 4: *The PYNQ processing overlay diagram. [Xilb]*

Hardware

The targeted hardware for this project is the Zynq processor on the PYNQ-Z1 board. However, this project is fully synthesizable and should be able to be targeted for any other Xilinx board.

Python and PYNQ

The PYNQ development board designed by Xilinx provides a Zynq chip which has an ARM CPU running at 650 MHz and an FPGA fabric that is programmable via an overlay [Xilb]. This performance allows for a linux operating system to be run on the CPU which in this case is Ubuntu, and hosts a Jupyter notebook to program and interface with the FPGA fabric using an overlay. This overlay contains mappings for ports and interfaces between the fabric and the CPU. This functionality is very unique in that both an ARM core and a fabric are on the same board. As shown by Fig. 4 the overlay sits between the processing system (CPU) and the programmable logic (FPGA). This overlay is loaded and programmed to the fabric through a Jupyter notebook and allows for native visualization and data interaction through any Python tools that work inside the IPython kernel. The overlay is represented by the yellow background with labels "Custom" and "Accelerator" and shows how the overlay is a communication layer between the processing system and the programmable logic.

It also contains a bitfile that will properly configure the FPGA [Xilc]. This bitfile is generated through the Vivado Design Suite that is provided by Xilinx by loading the output modules from the

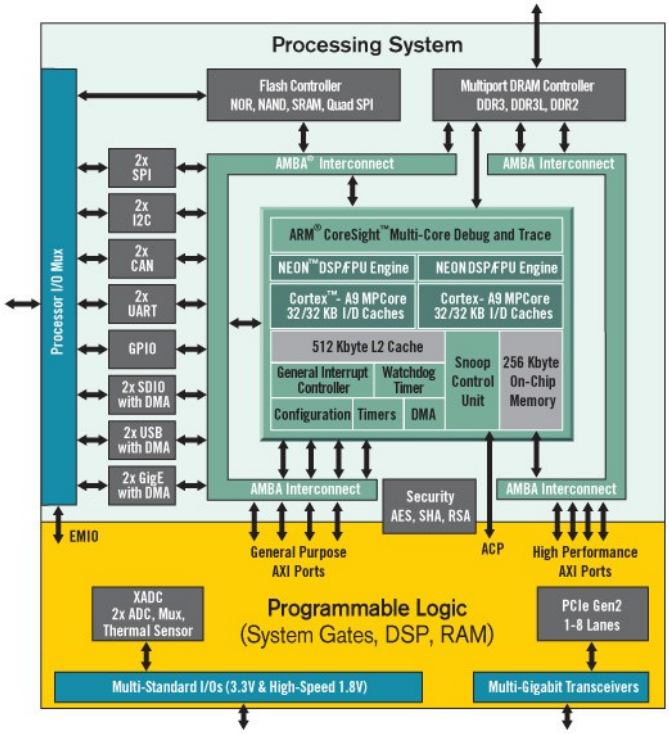


Fig. 5: *The PYNQ processing overlay diagram. [Xilb]*

caf-verilog module. A different bitfile must be created for every unique combination of configuration of the CAF and every device that is targeted. Every instantiation of the *CAF* Python class that has different parameters will require a new bitfile.

The Jupyter notebook itself is considered an interactive computing pool providing an interface to do computation and prototyping through a web browser. In this implementation it is meant to be an easier way for a non-hardware oriented person to be able to access a computational accelerator designed by a hardware engineer [Xilb].

A diagram of the processing and the programmable logic is shown in Fig. 5. The processor system is the Cortex-A9 processor that is running at 650MHz with 512MB of DDR3 RAM. The FPGA is a Zynq XC7020 part which has 13,300 logic slices, 53,200 6-input LUTs, 160,400 flip-flops, 630KB of block RAM, and 220 DSP slices. Later, a usage report is provided with a description of how the logic was optimized to make use of these primitives. It is possible to access the DRAM from the programmable logic (FPGA) through an AXI IP Core.

Software

Xilinx Vivado WebPack 2018.2

The Vivado design tool provides a simulator along with the ability to synthesize, elaborate, and implement the design [Xil14]. For this project, this built-in simulator was used exclusively. Other simulators were not chosen because the other target devices that this project seeks to be implemented on are likely to also be Xilinx products. The tool is free to download for anyone to use, and allows the hardware engineer to develop and synthesize HDL designs for Xilinx FPGA's. There is also a Software Development Kit that allows an engineer to write in C code. For this project, all modules are written in Verilog. This was done because of the need to instantiate multiple submodules that provide functionality

together. When running the synthesis tool, the output was very useful in helping make incremental design changes to fully optimize the board. Although none were used in this project, Xilinx does offer many free IP Cores that can be used in designs. They are black boxes that can be used in both simulation and the final implementation in HDL and block designs.

Python and Jupyter

This project made extensive use of the Python ecosystem through the use of pip, Jupyter, and many other packages. The reader is encouraged to view the `caf-verilog` source code [Sida] and view the releases that have been made on PyPI [Sidb]. When designing modules, a first test of what a signal should look like when operated on was done using the interactive plotting ability that is provided [Pro]. The generation of the modules was done using Jinja which provides both template parsing and rendering [Ron]. Whenever a simulated signal was changed, instead of having to write out a file or test bench by hand, a template was used to create the output and render it to the simulation directory. The signals that are used to create the signal generator were first quantized by using the NumPy library and then written to a file that gets used a memory buffer in the signal generator [Num]. Most of the mathematical operations that are implemented were first verified using this library. This project requires the use of orthogonal signals to ensure that the spectral density that is being tested is isolated from the others. This was possible using the gps-helper module that implements the GPS gold codes that are orthogonal PRN sequences [WSa].

Quantization

In order to use a signal in the digital domain, a signal must first be quantized by an analog to digital converter (ADC). Most ADC's that are available are able to provide a 12-bit value, and some newer devices are now able to provide 16-bits [Ana]. However, for this project 12-bit signed signals were used during testing as this is a very nice number to compute mentally and still provides minimal energy loss when plotting on the spectrum.

Inspecting signals after quantization is important because when signals are reduced in size there is information loss. This is demonstrated by Fig. 6 where a 12 bit and 8 bit quantization of a cosine signal is shown. Quantization helper functions are provided in caf_verilog with the help of scikit-dsp-comm's simpleQuant function [WSb]. This means that the full bit value of the signal cannot be used otherwise there is signal loss to DC gain. The signals must be equal over 0. For a 12-bit quantization of a vector for example the numbers must be in the range (-4095, 4095) in comparison to the two's complement full value of (-4096, 4095). This is all necessary because the computation that is done on the FPGA will be done using fixed point or an integer value. This also reduces power and cost on the FPGA [FR]. Test files are written out and read back as integer values via this module by all the other classes for tests and verification.

Complex Multiply

As an example for why this module is necessary, an example of frequency shifting a signal is presented. In Fig. 7 we have two inputs: a positive frequency signal on top, and a negative frequency signal in the middle. The output is shown in the bottom plot. All of these signals are shown in a spectral density plot, with both sampling frequencies normalized to a value of 1 for presentation. What we see is that the resulting spectrum has a

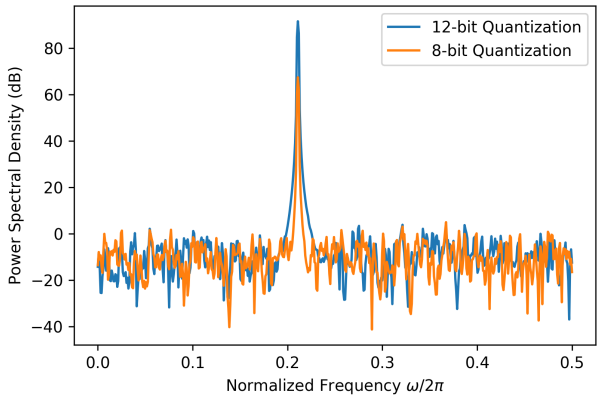


Fig. 6: *12-bit and 8-bit Quantization Comparison.*

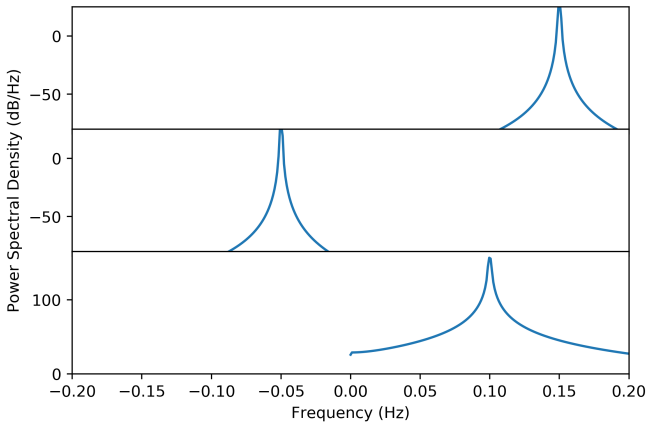


Fig. 7: *Inputs (top and middle) and output (bottom) of the CPX Multiply Verilog module.*

signal at a frequency of the sum of the two negative and positive frequency signals. This is what is expected. This method is what is used to shift the captured signal for the CAF.

Signed multiplication in Verilog can be done by specifying the signed data type. Any multiply of two numbers of the same size requires twice the number of bits in the result [Tum]. However, in this project the need for different size operands arises. This module takes in two complex numbers and performs a pipelined multiplication on the data. Before the result is provided to the master, the result is truncated. It should be noted that no timing constraint violations were encountered during the implementation. The only timing constraint that was provided was the slew rate for the fabric clock, and all other constraints were Vivado defaults.

The specific pipeline steps are presented in Table 1 which shows which operations are completed in which pipeline stage. Stages 1 and 2 are always conditionally assigned based on the current state of the AXI interface so that resources are not constantly being used. This helps for timing and for power usage. The result is then truncated and returned to the master when the master's ready signal is asserted. Because this is a pipelined implementation, an input and output can be processed every clock.

A code listing of the Verilog HDL output is provided as reference. The two blocks that are shown are for the first step

TABLE 1: *CPX Multiply Stages*

Stage	Operation
1	xi * yi
1	xq * yq
1	xi * yq
1	xq * yi
2	xu - yv
2	xv + yu
3	Truncate

through the third step. The first two steps can be seen to only be calculated when the master signal conditions are correct.

```
always @(posedge clk) begin
   if (m_axis_tvalid & s_axis_tready) begin
      xu <= xi * yi;
      yv <= xq * yq;
      xv <= xi * yq;
      yu <= xq * yi;
   end else begin
      xu <= xu;
      yv <= yv;
      xv <= xv;
      yu <= yu;
   end // else: !if(m_axis_tvalid & s_axis_tready)
end

always @(posedge clk) begin
   if(m_axis_tready) begin
      xu_out <= xu;
      yv_out <= yv;
      xv_out <= xv;
      yu_out <= yu;
      i_sub <= xu_out - yv_out;
      i_sub_out <= i_sub;
      q_add <= xv_out + yu_out;
      q_add_out <= q_add;
   end // if (m_axis_tvalid)
   else begin
      xu_out <= xu_out;
      yv_out <= yv_out;
      xv <= xv;
      xv_out <= xv_out;
      yu <= yu;
      yu_out <= yu_out;
      i_sub <= i_sub;
      i_sub_out <= i_sub_out;
      q_add <= q_add;
      q_add_out <= q_add_out;
   end // else: !if(m_axis_tvalid)
   i <= i_sub_out[xi_bits+yi_bits-1:
                  xi_bits+yi_bits-i_bits];
   q <= q_add_out[xq_bits+yq_bits-1:
                  xq_bits+yq_bits-q_bits];
end // always @ (posedge clk)
```

Signal Generator

The signal generator module is implemented using a half sine lookup table and accumulator. This is commonly known as a numerically controlled oscillator in direct digital synthesis [MS]. This module produces a sine wave at the specified frequency by using a modulo counter that increments a phase value at every clock cycle. Note that the sampling frequency of the signal, 625kHz, is different from the clock frequency of the board, at 250MHz. The number of phase bits that are necessary are determined by the sampling frequency and the frequency resolution specified by Eq. 1.

$$\left\lceil \log_2\left(\frac{f_{\text{clk}}}{\text{freq\_res}}\right)\right\rceil = \text{phase\_bits} \tag{1}$$

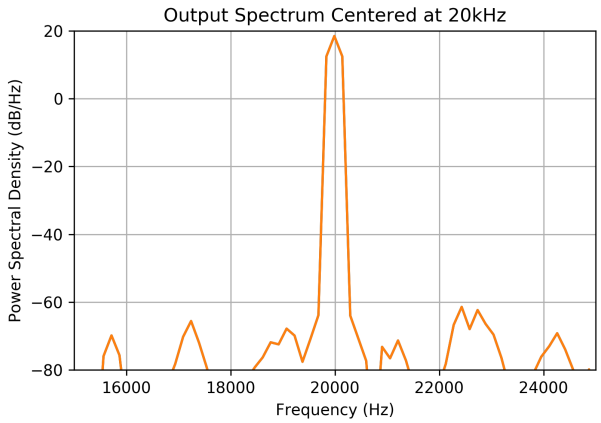


Fig. 8: *Cosine centered at 20kHz with 8-bits and 200Hz resolution.*

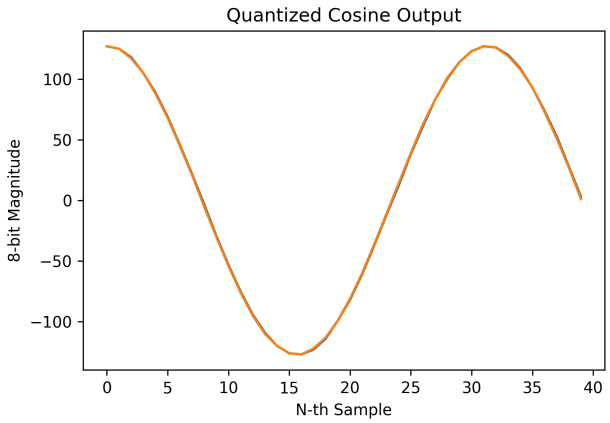


Fig. 9: *Cosine centered at 20kHz with 8-bits and 100Hz resolution.*

The output of one cycle is shown in Fig. 9. The values that are supplied to the module for the lookup table are generated using the NumPy sine function and are quantized using helper methods included in the caf_verilog module. To set the frequency of the signal generator, a phase step or increment value must be provided by Eq. 2.

$$\frac{f_{\text{out}} \cdot 2^{\text{phase\_bits}}}{f_{\text{clk}}} = \text{phase\_increment} \tag{2}$$

An example spectrum of the output of the signal generator that is created from the Python class is shown in Fig. 8. While no calculation of power has been provided, a parameter n_bits sets the signal strength. For this project, a value of 8-bits was found to be sufficient to provide a frequency shifted signal. The same settings used to generate the module used as an example in this section are used in Fig. 10 by using the *SigGen* class.

Frequency Shift

The frequency shift module takes in the same parameters as the signal generator module and adds an input for a complex value to shift. This module needs to make sure that different bit width signals are multiplied together correctly and that the pipeline is managed correctly to ensure that there are no phase shifts. Fig. 10 shows an input signal, and the resulting shifted signal. When using the Python generated Verilog module, a negative value for the frequency will be taken care of by setting a bit in the

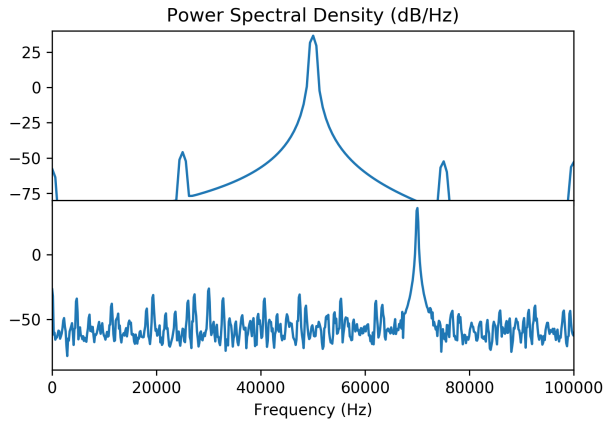


Fig. 10: *Input signal at 50kHz (top), and output signal at 70kHz (bottom).*

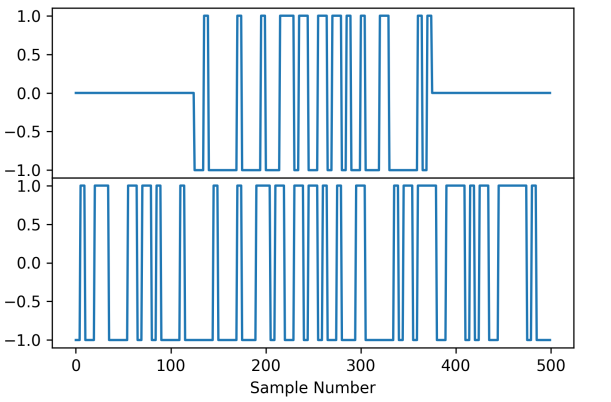


Fig. 11: *A reference and received signal (top and bottom) simulated from a PRN sequence from gps-helper.*

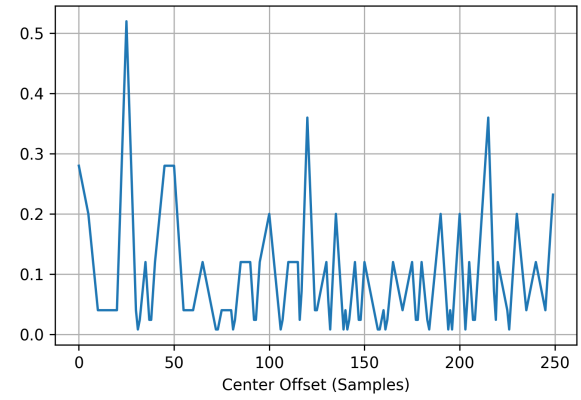


Fig. 12: *The cross correlation of two simulated signals, showing a positive offset of 25 samples using xcorr.*

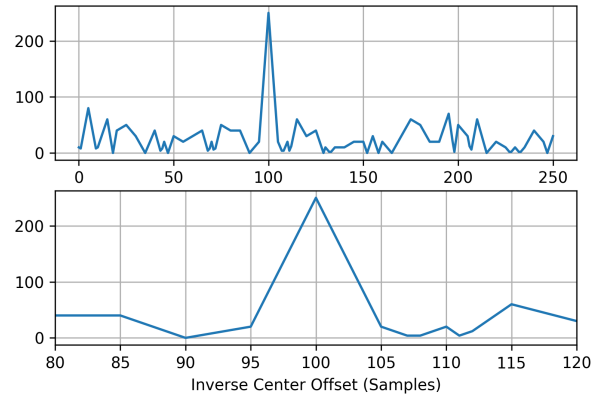


Fig. 13: *Cross correlated output of a signal of length 250.*

module parameters to perform the complex conjugate on the signal generator output.

Required inputs for the *FreqShift* Python class are the input vector 'x', and the number of bits for the I and Q data that it represents. The same parameters are passed to the *FreqShift* class so that the *SigGen* module can be instantiated internally and accessed for naming by the Jinja template for the module.

Cross Correlation

The cross correlation is useful in comparing the time offset between two signals. As an example, a pseudorandom sequence signal provided by gps-helper [WSa] is time shifted in Fig. 11 by ten samples. Both of these signals are a non-return to zero representation of the binary bit sequence. The reference is shown with zero padding on either end so the visual representation stays centered between the two signals.

The general form of the cross correlation is [ZT08]:

$$(f \star g)(\tau) = \int_{-\infty}^{\infty} \overline{f(t)} g(t+\tau)\, dt \tag{3}$$

In Eq. 3 the signal $f(t)$ is shown with the complex conjugate, and the signal $g(t)$ is shown with a time or sample shift of τ. When translated into the discrete form, the form looks like the following:

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m] g[m+n] \tag{4}$$

When looking at the cross product in discrete form (4), it is possible to see that the form of the multiplication and addition closely follows that of the dot product (Eq. 5).

$$a \cdot b = \sum_{i=1}^{n} a_i * b_i \tag{5}$$

These series of equations provide a means of determining where the signals are correlated in time, or if they are orthogonal meaning they are not correlated at all [ZT08]. In order to capture the full signal power with the dot product, it is necessary to store twice the length of the reference signal for correlation as shown in Fig. 11. Furthermore, as compared to Fig. 12 which uses xcorr, the dot product method that produced Fig. 13 has a much higher magnitude. This is because the xcorr method uses an FFT, and the result is normalized to one. We also note that the axis for samples is denoted as an inverse offset. When the peak is generated with dot products, the center is going to be the center of the sample length. This is because the multiply and accumulate has the highest magnitude in the center as compared to the xcorr method with FFT's which produces a normalized axis around zero. Since we are always looking at what offset is necessary to cause the shift

back to the reference, it is left as a sample offset. This also makes verifying the Verilog test benches much more straight forward. However, with the dot product, the magnitude that is achieved when a full correlation is hit is the length of the correlation sequence itself. This means that a longer integration time allows for a higher fidelity difference between signal magnitudes of surrounding shifted correlations. This method reduces the amount of multiplies that are necessary and is much simpler to implement on an FPGA. These results were verified using the xcorr function from scikit-dsp-comm and the dot product function provided by NumPy. The simulation for this function required the output of the entire sequence to be written to and sequentially read from disk. When running the simulations it was found that the very last dot product in the sequence was missing. A full cross correlation using the dot product actually has two times the length plus one to account for both positive and negative offsets.

Dot Product

The final CAF solution uses a pipelined multiply and accumulate. When the implementation was run, it was found that a pipelined implementation was able to make use of the primitive DSP48 type. Further fine-tuning suggestions were taken to ensure that the multiply and accumulate functionality of the primitive type was taken advantage of correctly [FWS].

ArgMax

Because there is a need to compare the magnitudes of complex numbers, the argmax function is required. The mathematical absolute value of a complex number is described in Eq. 6. However, finding the true absolute value of the number requires the implementation of the square root. The first option that was looked at was a binary square root algorithm [Min13] that only uses base 2 division. However, this can take a variable amount of clock cycles. An implementation is provided in the sqrt package as reference. The other option is the CORDIC logic core provided by Xilinx which also would apply backpressure [Xila], essentially sequentially buffering the result by a fixed number of clocks.

$$r = \sqrt{x^2 + y^2} \tag{6}$$

After comparing results and performing the argmax using these different methods a decision was made to just use the squares of each of the real and imaginary components. This is possible because we can use the proportion of the squared values and their square roots to compute the argmax with the same result. Since the largest magnitude squared value is made up of both a real and imaginary component, it is enough to say that the largest magnitude $(x^2 + y^2)$ will be sufficient. The result is provided back by the next clock, with only a delay in the pipeline for the first multiply. Then, comparisons are done within the module itself to find the max. This also allows for taking advantage of the larger integration time by allowing larger max values to propagate through. The trade-off is that there is much larger utilization with multiple instantiations all growing in size as the multiply operands increase in size. Inspecting the utilization of the synthesized and implemented designs did not seem to indicate that this was the limiting factor in the design layout growth.

CAF Module

The CAF module uses a generate variable, which is part of the Verilog standard [ver01] to implement the frequency shifts and corresponding cross correlations. A reference buffer and a capture buffer are instantiated in this module that provide the input to the pipeline as shown in Fig. 14. This module is a slave to a master as it is being driven by the data lines.

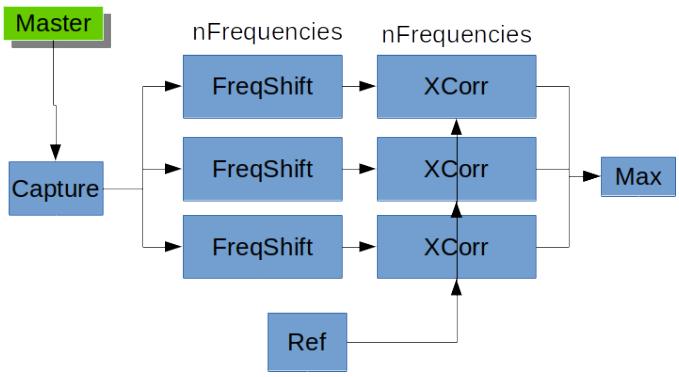


Fig. 14: *Block diagram of a CAF implementation with three frequencies.*

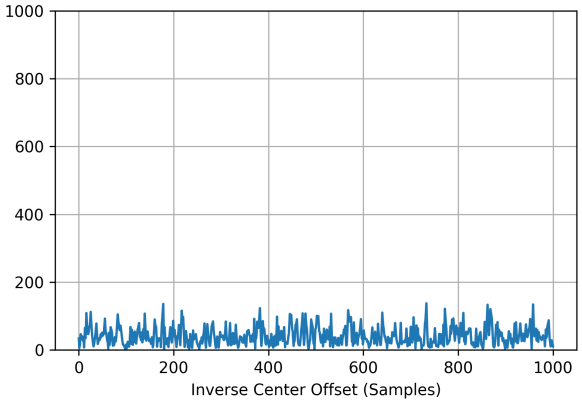


Fig. 15: *Cross correlation of a frequency shifted signal.*

The results of a frequency shifted correlation is shown in Fig. 15, and an autocorrelation is shown in Fig. 16. We see that in Fig. 15 there is no peak. This is because two orthogonal signals should not have any correlation energy.

In the next code listing, the Python class definition for *CAF*

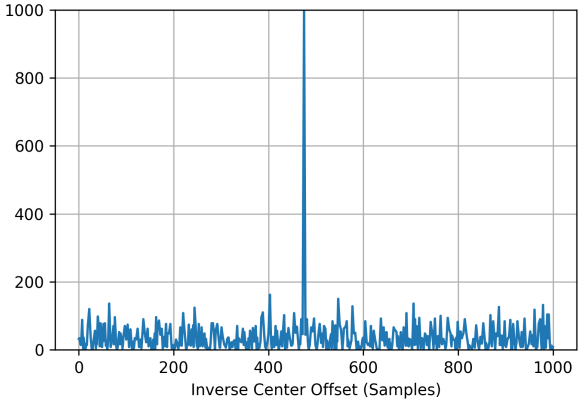


Fig. 16: *Autocorrelation output with length of implemented design.*

TABLE 2: 12 phase bits, 8-bit multiplication, 49 frequencies, and 1000 samples.

Resource	Utilization	Available	Utilization %
LUT	32682	53200	61.43
LUTRAM	490	17400	2.82
FF	28695	106400	26.97
BRAM	25.50	140	18.21
DSP	196	220	89.09
IO	53	125	42.40

is provided for reference. The class takes in both a reference and received or captured signal, and the number of bits requested to represent the signals. These two signals are required parameters. The reference signal is used to produce a stored reference as a capture buffer module, and the received signal is used in the generated test bench. The same parameters for the SigGen and FreqShift modules are required here as well, as they are passed down to their instantiations for the CAF to instantiate.

```
class CAF(CafVerilogBase):

    def __init__(self, reference, received, foas,
                 ref_i_bits=12, ref_q_bits=0,
                 rec_i_bits=12, rec_q_bits=0,
                 fs=625e3, n_bits=8,
                 pipeline=True, output_dir='.'):
```

Synthesis and Implementation

Both the synthesis and implementation were completed successfully, and all timing constraints were met by the tool. Several different design sizes were elaborated and implemented, all ending up with different utilization amounts. The final design iteration that was able to maximize the iteration time is described by Table 2. Each of these tables describes a different usage that is still below the specific size of the Pynq board. For different devices, new *CAF* Python class instantiations should be used to explore board usages by using the Verilog module outputs to follow the Vivado design process.

The final implementation run shown by Table 2 was able to use the most of the resources of the board evenly because of the 8-bit multiplication [FWS]. The first couple implementations were using 12-bit numbers because that was what was nominally chosen for the simulations. However, since regenerating the module is very simple, a new CAF module was written out using the module and tested with different shifts. The final implementation has 49 different frequency offsets and an integration sample length of 1000.

Future Work and Enhancements

When the original implementation of the sin and cosine generator was created, a half-sine method was used. While functionally sound, it is possible to decrease size by using a quarter-sine implementation where only a fourth of the sine is stored [Tec].

While the body of work for caf-verilog supports the modeling of the caf itself, this project can be used as a basis for incorporating Verilog as an extension to the wider scientific computing field. The SymPy Development Team has already made significant contributions in this realm, and is being used in many projects to support code generation for various languages such as c, c++, and Julia [Sym]. Using a common API, it should be possible to also provide an extension to incorporate the Verilog HDL.

REFERENCES

[Na18] National Executive Committee for Space-Based Positioning, Navigation, and Timing . Gps.gov technical documentation, 2018. URL: https://www.gps.gov/technical/#prn.

[Ana] Analog Devices. Ad7903 datasheet. URL: https://www.analog.com/en/products/ad7903.html.

[Arm17] Arm Limited. Amba axi and ace protocol specification documentation, 2017. URL: https://developer.arm.com/docs/ihi0022/latest.

[FR] Ambrose Finnerty and Herve Ratigner. Reduce power and cost by converting from floating point to fixed point. URL: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf.

[FWS] Yao Fu, Ephrem Wu, and Ashish Sirasao. 8-bit dot-product acceleration. URL: http://xilinx.com.

[Har05] Glenn D. Hartwell. Improved geo-spatial resolution using a modified approach to the complex ambiguity function (caf), 2005. URL: https://calhoun.nps.edu/handle/10945/2033.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[KPK81] W. C. Knight, R. G. Pridham, and S. M. Kay. Digital signal processing for sonar. *Proceedings of the IEEE*, 69(11):1451–1506, Nov 1981. doi:10.1109/PROC.1981.12186.

[Min13] Kwa Tak Ming. *The Fundamental Operations in Bead Arithmetic - How to Use the Chinese Abacus*. France Press, 2013.

[MS] Eva Murphy and Colm Slattery. Ask the application engineer—33: All about direct digital synthesis. URL: https://www.analog.com/en/analog-dialogue/articles/all-about-direct-digital-synthesis.html.

[Num] NumPy Developers. Numpy. URL: http://www.numpy.org/.

[Pro] Project Jupyter. Project jupyter. URL: https://jupyter.org.

[Ron] Armin Ronacher. Jinja 2 (the python template engine). URL: http://jinja.pocoo.org/.

[Sida] Chiranth Siddappa. Caf verilog. URL: https://github.com/chiranthsiddappa/caf_verilog.

[Sidb] Chiranth Siddappa. caf-verilog. URL: https://pypi.org/project/caf-verilog/.

[Ste81] S. Stein. Algorithms for ambiguity function processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(3):588–599, June 1981. doi:10.1109/TASSP.1981.1163621.

[Sym] SymPy Development Team. Codegen - sympy documentation. URL: https://docs.sympy.org/latest/modules/utilities/codegen.html.

[Tec] Gissequalt Technologies. Building a quarter sine-wave lookup table. URL: https://zipcpu.com/dsp/2017/08/26/quarterwave.html.

[Tum] Greg Tumbush. Signed arithmetic in verilog 2001 – opportunities and hazards. URL: http://www.tumbush.com/published_papers/.

[ver01] Ieee standard verilog hardware description language. *IEEE Std 1364-2001*, pages 1–792, Sep. 2001. doi:10.1109/IEEESTD.2001.93352.

[Wei94] L. G. Weiss. Wavelets and wideband correlation processing. *IEEE Signal Processing Magazine*, 11(1):13–32, Jan 1994. doi:10.1109/79.252866.

[WSa] Mark Wickert and Chiranth Siddappa. Gps helper. URL: https://gps-helper.readthedocs.io/en/latest/?badge=latest.

[WSb] Mark Wickert and Chiranth Siddappa. Scikit dsp comm. URL: https://scikit-dsp-comm.readthedocs.io/en/latest/digitalcom.html.

[Xila] Xilinx Inc. Cordic v6.0. URL: https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf.

[Xilb] Xilinx Inc. Python productivity for zync (pynq). URL: https://pynq.readthedocs.io/en/v2.3/.

[Xilc] Xilinx Inc. Running the generate programming file process for fpgas. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_p_generate_fpga_programming_file.htm.

[Xil14] Xilinx Inc. Vivado design suite user guide: Design flows overview, 2014. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug892-vivado-design-flows-overview.pdf.

[ZT08] Rodger E. Ziemer and William H. Tranter. *Principles of Communications*. Wiley Publishing, 6th edition, 2008.

Developing a Graph Convolution-Based Analysis Pipeline for Multi-Modal Neuroimage Data: An Application to Parkinson's Disease

Christian McDaniel‡, Shannon Quinn, PhD‡*

Abstract—Parkinson's disease (PD) is a highly prevalent neurodegenerative condition originating in subcortical areas of the brain and resulting in progressively worsening motor, cognitive, and psychiatric (e.g., depression) symptoms. Neuroimage data is an attractive research tool given the neurophysiological origins of the disease. Despite insights potentially available in magnetic resonance imaging (MRI) data, developing sound analytical techniques for this data has proven difficult. Principally, multiple image modalities are needed to compile the most accurate view possible; the process of incorporating multiple image modalities into a single holistic model is both poorly defined and extremely challenging. In this paper, we address these issues through the proposition of a novel graph-based convolutional neural network (GCN) architecture and present an end-to-end pipeline for preprocessing, formatting, and analyzing multimodal neuroimage data. We employ our pipeline on data downloaded from the Parkinson's Progression Markers Initiative (PPMI) database. Our GCN model outperforms baseline models, and uniquely allows for direct interpretation of its results.

Introduction

Affecting more than 1% of the United States population over the age of 60, Parkinson's disease (PD) is the second-most prevalent age-related neurodegenerative disease following Alzheimer's disease [RST14]. PD diagnosis has traditionally relied on clinical assessments with some degree of subjectivity [GGL+18], often missing early-stage PD altogether [DDH16]. Benchmarks for delineating PD progression or differentiating between similar conditions are lacking [[LMS+18], [LWX+12]]. As such, many efforts have emerged to identify quantitatively rigorous methods through which to distinguish PD.

Neuroimage data is an attractive tool for PD research. Magnetic resonance imaging (MRI) in particular is safe for patients, highly diverse in what it can capture, and decreasing in cost to acquire. Recent work shows that multiple MRI modalities are required to provide researchers and clinicians with the most accurate view of a patient's physiological state [[LCL+15], [DDH16], [LWX+12]]. For example, anatomical MRI (aMRI[1]) data is useful for identifying specific brain regions, but the Euclidean distance between regions does not well-approximate the functional or structural connectivity between them. Diffusion-weighted MRI (dMRI) measures the flow of water through the brain in order to track the tube-like connections between regions (i.e., tracking *nerve fiber bundles* a.k.a. *tracts* via white matter *tractography*; see Appendix A in the `appendices` file on our GitHub repository[2] for more information), and functional MRI (fMRI) measures changes in blood oxygenation throughout the brain over time to approximate which regions of the brain function together. As such, it is useful to analyze a combination of these modalities to gain insights from multiple measures of brain physiology. Processing and analyzing multi-modal data together is both poorly defined and extremely challenging, requiring combined expertise from neuroscience and data analytics.

MRI data is inherently noisy data and requires extensive preprocessing before analysis can be performed. This is often left to the researcher to carry out; many techniques exist, and the technical implementation decisions made along the way can affect the outcome of later analysis. This is a major barrier to reproducibility and prevents data analysts from applying their skills in this domain. More work is needed to automate the procedure and provide better documentation for steps requiring case-specific input. To that end, we discuss our findings and methods below, and our code is available on GitHub.

Following preprocessing, we address the issue of analyzing multimodal MRI data together. Previous work has shown that graph-based signal processing techniques allow multimodal analysis in a common data space [[DMF+17], [KPF+18], [ZHC18]]. It has been shown that graph-based signal processing classifiers can be incorporated in neural network-like architectures and applied to neuroimage data. Similar to convolutional neural networks, Graph Convolutional Networks (GCNs) learn *filters* over a graph so as to identify patterns in the graph structure, and ultimately perform classification on the nodes of the graph. In this paper, following the discussion of our preprocessing pipeline, we propose a novel GCN architecture which uses graph attention network (GAT)

‡ *University of Georgia*
* *Corresponding author: spq@uga.edu*

1. We use "anatomical MRI" to refer to standard *T1-weighted* (T1w) MR imaging. "T1 weighted" refers to the specific sequence of magnetic and radio frequency pulses used during imaging. T1w MRI is a common MR imaging procedure and yields high-resolution images; different tissues and brain regions can be distinguished.

2. https://github.com/xtianmcd/GCNeuro

layers to perform whole-graph classification on graphs formed from multimodal neuroimage data.

On data downloaded from the Parkinson's Progression Markers Initiative (PPMI), we compare the performance of the novel GCN architecture to that of baseline models. We find that our GCN model outperforms baseline models on our data. The weights from GAT layers provide a means for direct interpretation of the results, indicating which brain regions contributed the most to the distinction between patients with PD and healthy controls.

Related Works

While genetic and molecular biomarkers have exhibited some efficacy in developing a PD blueprint [[GGL+18], [MLL+18], [BP14]], many research efforts have turned to neuroimaging due to its noninvasive nature and alignment with existing knowledge of the disease. Namely, PD affects a major dopamine-producing pathway (i.e., the nigrostriatal dopaminergic pathway) of the brain [Bro16], and results in various structural and functional brain abnormalities that can be captured by existing imaging modalities [[ZYH+18], [MLL+18], [GLH+14], [TBvE+15], [LSC+14], [GRS+16]]. Subsequent whole-brain neuroimage analysis has identified PD-related regions of interest (ROIs) throughout the brain, from cortical and limbic regions to the brainstem and cerebellum [[BWS+11], [TBvE+15], [GRS+16]].

As neuroimage data has accumulated, researchers have worked to develop sound analytical techniques for the complex images. Powerful machine learning techniques have been employed for analyzing neuroimage data [[MLL+18], [TBvE+15], [BWS+11], [LSC+14]], but algorithmic differences can result in vastly different results [[GLH+14], [Kum18], [ZYH+18]]. [CJM+17] and [GRS+16] found that implementation choices made during the processing pipeline can affect analysis results as much as anatomical differences themselves (e.g., when performing white matter tractography on diffusion-weighted MRI (dMRI) data and in group analysis of resting-state functional MRI (rfMRI) data, respectively). To overcome the effect of assumptions made by a given analysis algorithm, many researchers have turned to applications of deep machine learning (DL) for neuroimage data analysis. Considered "universal function approximators" [HKK90], DL algorithms are highly flexible and therefore have low bias in their modeling behavior. Examples of DL applications to neuroimage analysis are widespread. [KUH+16] proposes a 3D convolutional neural network (CNN) for skull stripping 3D brain images, [HDC+18] proposes a novel recurrent neural network plus independent component analysis (RNN-ICA) model for fMRI analysis, and [HCS+14] demonstrate the efficacy of the restricted Boltzmann machine (RBM) for network identification. [LZC17] offer a comprehensive review of deep learning-based methods for medical image computing.

Multi-modal neuroimage analysis is increasing in prevalence [[BSS+18], [LCL+15], [DDH16], [LMS+18], [LWX+12]] due to limitations of single modalities, resulting in larger and increasingly complex data sets. Recently, researchers have utilized advances in graph convolutional networks to address these concerns. We discuss the mathematical background of graph convolutional networks (GCNs) and graph attention networks (GATs, a variant of GCNs with added attention mechanisms) in the Methods Section below and Appendix B in the `appendices` file on GitHub. Principally, our model is based on advancements made by [KW217] and [VCC18] on GCNs and GATs, respectively.

This work follows from previous efforts applying GCNs to similar classification tasks. [SNF+13] - in addition to providing in-depth intuition behind spectral graph processing (i.e., processing a signal defined on a graph structure) - demonstrate spectral graph processing on diffusion signals defined on a graph of connected brain regions. Their paper preceded but laid the groundwork for incorporating spectral graph processing into convolutional neural network architectures. To classify image objects based on multiple "views" or angles, [[KZS15], [KCR16]] developed "siamese" and "multi-view" neural networks. These architectures share weights across parallel neural networks to incorporate each view of the data. They group examples into pairs, aiming to classify the pairs as being from the same class or different classes.

Efforts to utilize GCNs for multimodal neuroimage data have used similar pairwise grouping as a way to increase the size of their data set. [[DMF+17], [KPF+18]] train GCN models to learn similarity metrics between subjects with Autism Spectrum Disorder (ASD) and healthy controls (HC), using fMRI data from the Autism Brain Imaging Data Exchange (ABIDE) database. [ZHC18] apply a similar architecture to learn similarity metrics between subjects with PD and HC, using dMRI data from the PPMI data set. Their work inspired our paper; to our knowledge, we are the first publication that uses GCNs to predict the class of neuroimage data directly, instead of making predictions on pairwise examples.

Discussion of the Processing Pipeline

This section walks through our pipeline, which handles the formatting and preprocessing of multimodal neuroimage data and readies it for analysis via our GCN architecture. We reference the specific python files that handle each task, and we provide some background information. More information can be found in the Appendices on GitHub.

Data Formatting

MRI signals are acquired through the application of precisely coordinated magnetic fields and radiofrequency (RF) pulses. Each image is reconstructed from a series of recordings averaged over many individual signals, and requires extensive artifact correction and removal before it can be used. This inherently results in noisy measurements, magnetic-based artifacts, and artifacts from human error such as motion artifacts [[Wan15], [HBL10]]. As such, extensive preprocessing must be performed to clean the data before analysis. Appendix A on our GitHub page provides more details on the main MRI modalities.

Our pipeline assumes that a "multi-zip" download is used to get data from the PPMI database[3]. The file `neuro_format.py` combines the data from multiple download folders into a single folder, consolidating the multiple zip files and recombining data from the same subject.

Next, before preprocessing, images should be converted to the Neuroimaging Informatics Technology Initiative (NIfTI)[4] file format. Whereas many MRI data are initially in the Digital Information and Communications in Medicine (DICOM)[5] format for standardized transfer of medical data and metadata, the NIfTI format is structured for ease of use when conducting computational analysis and processing on these files. The size, orientation, and location in space of the voxel data is dependent on settings

3. The "Advanced Download" option on the PPMI database splits the data into multiple zip files, separating files from the same subject.

used during image acquisition and requires an *affine matrix* to relate two images in a standard coordinate space. The NIfTI file format automatically associates each image with an affine matrix as well as a *header file*, which contains other helpful metadata. The software `dcm2niix`[6] is helpful for converting the data from DICOM format to NIfTI format.

Next, it is common practice to convert your data file structure to the Brain Imaging Data Structure (BIDS)[7] format. Converting data to the BIDS format is required by certain softwares, and ensures a standardized and intuitive file structure. There exist some readily available programs for doing this, but we wrote our own function specifically for PPMI data in `make_bids.py`, as the PPMI data structure is quite nuanced. This file also calls `dcm2niix` to convert the image files to NIfTI format.

Data Preprocessing

This subsection discusses the various softwares and commands used to preprocess the multimodal MRI data. The bash script `setup` should help with getting the necessary dependencies installed[8] . The script was written for setting up a Google cloud virtual machine, and assumes the data and pipeline files are already stored in a Google cloud bucket.

The standard software for preprocessing anatomical MRI (aMRI) data is Freesurfer[9]. Although an actively developed software with responsive technical support and rich forums, receiving training for Freesurfer may still be helpful. The `recon-all` command performs all the steps needed for standard aMRI preprocessing, including motion correction, registration to a common coordinate space using the Talairach atlas by default, intensity correction and thresholding, skull-stripping, region segmentation, surface tessellation and reconstruction, statistical compilation, etc.

The entire process takes around 15 or more hours per image. Support for GPU-enabled processing was stopped years ago, and the `-openmp <num_cores>` command, which allows parallel processing across the designated number of cores, may only reduce the processing time to around 8-10 hours per image[10]. We found that running parallel single-core CPU processes worked the best, especially when many processing cores are available. For this we employed a Google Cloud Platform virtual machine and utilized the python module `joblib.Parallel` to run many single-core processes in parallel. For segmentation, the Deskian/Killiany atlas is used, resulting in around 115 volume segmentations per image, to be used as the nodes for the graph.

The Functional Magnetic Resonance Imaging of the Brain (FMRIB) Software Library (FSL)[11] is often used to preprocess diffusion data (dMRI). The *b0* volume is taken at the beginning of dMRI acquisition and is used to align dMRI images to aMRI images of the same subject. This volume is isolated (`fslroi`) and merged with b0's of other clinic visits (CVs)[12] for the same subject (`fslmerge`). `fslmerge` requires that all dMRI acquisitions for a given subject have the same number of coordinates (e.g., (116,116, **78** ,65) vs. the standard (116,116, **72** ,65)). Since some acquisitions had excess coordinates, we manually examined these images and, if possible, removed empty space above or below the brain. Otherwise, these acquisitions were discarded. Next, the brain is isolated from the skull (*skull stripped*, `bet` with the help of `fslmaths -Tmean`), magnetic susceptibility correction is performed *for specific cases* (see below) using `topup`, and eddy correction is performed using `eddy_openmp`. Magnetic susceptibility and eddy correction refer to specific noise artifacts that significantly affect dMRI data.

The `topup` tool requires two or more dMRI acquisitions for a given subject, where the image acquisition parameters `TotalReadoutTime` and/or `PhaseEncodingDirection` (found in the image's header file) differ from one another. Since the multiple acquisitions for a given subject typically span different visits to the clinic, the same parameters are often used and `topup` cannot be utilized. We found another software, BrainSuite[13], which can perform magnetic susceptibility correction using a single acquisition. Although we still include FSL in our pipeline since it is the standard software used in many other papers, we employ the BrainSuite software's Brain Diffusion Pipeline to perform magnetic susceptibility correction and to align the corrected dMRI data to the aMRI data for a given subject (i.e., *coregistration*).

First, a BrainSuite compatible brain mask is obtained using `bse`. Next, `bfc` is used for bias field (magnetic susceptibility) correction, and finally `bdp` performs co-registration of the diffusion data to the aMRI image of the same subject. The calls to the Freesurfer, FSL, and BrainSuite software libraries are included in `automate_preproc.py`.

Once the data has been cleaned, additional processing is performed on the diffusion (dMRI) data. As discussed in the Introduction section, dMRI data measures the diffusion of water throughout the brain. The flow of water is constricted along the tube-like pathways (tracts) that connect regions of the brain, and the direction of diffusion can be traced from voxel to voxel to approximate the paths of tracts between brain regions. There are many algorithms and softwares that perform tractography, and the choice of algorithm can greatly affect the analysis results. We use the Diffusion Toolkit (DTK)[14] to perform multiple tractography algorithms on each diffusion image. In `dtk.py` we employ four different diffusion tensor imaging (DTI)-based deterministic tractography algorithms: Fiber Assignment by Continuous Tracking (FACT; [MCCv99]), the second-order Runge–Kutta method (RK2; [BPP+00]), the tensorline method (TL; [LWT+03]), and the interpolated streamline method (SL, [CLC99]). [ZZW+15] provide more information on each method. `dti_recon` first transforms the output file from Brainsuite into a usable format for DTK, and then `dti_tracker` is called for each of the tractography algorithms. Finally, `spline_filter` is used to smooth the generated tracts, denoising the outputs. Now that

4. https://nifti.nimh.nih.gov
5. https://www.dicomlibrary.com
6. https://github.com/rordenlab/dcm2niix
7. https://bids.neuroimaging.io
8. We install the softwares to the home (~) to avoid permission issues during remote Google cloud session. Several environment variables used by Freesurfer need to be hard coded to accommodate this download location. See the *setup* bash script provided for details.
9. https://surfer.nmr.mgh.harvard.edu
10. In the release notes, it is recommended for multi-subject pipelines to use a single core per image and process subjects in parallel; we also found this to provide the greatest speedup. Multiprocessing only reduces the processing time by a few hours, so parallelization is more important. We did not use GPUs; the time required to transfer data on and off GPU cores may diminish the speedup provided by GPU processing. Also, Freesurfer has not supported GPUs for quite some time, and we were unable to compile Freesurfer to use newer versions of CUDA.
11. https://fsl.fmrib.ox.ac.uk/fsl/fslwiki
12. We use "clinic visit" or CV to refer to the MRI acquisitions (anatomical and diffusion) obtained during a single visit to the clinic.
13. http://brainsuite.org

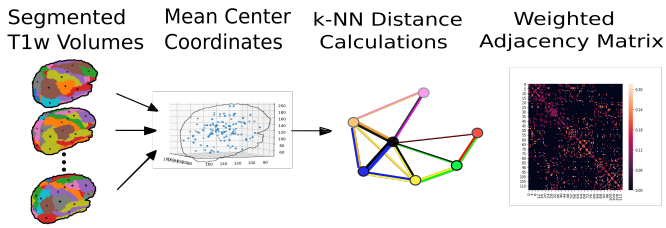


Fig. 1: *A depiction of the steps involved in forming the adjacency matrix. First, anatomical images are segmented into regions of interest (ROIs), which represent the vertices of the graph. The center voxel for each ROI is then calculated. An edge is placed between each node* i *and its* k*-nearest neighbors, calculated using the center coordinates. Lastly, each edge is weighted by the normalized distance between each node* i *and its connected neighbor* j*.*

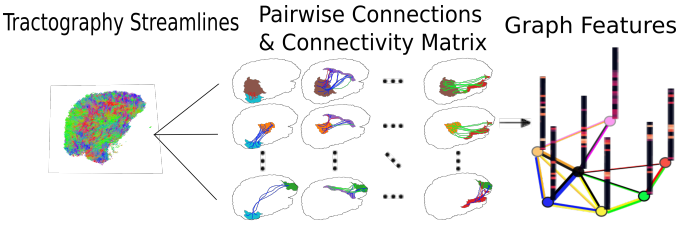


Fig. 2: *The process of generating the features from a single tractography algorithm is shown. Tractography streamlines are aligned to a corresponding anatomical image. The number of streamlines connecting each pair of brain regions is calculated to represent the strength of connection. Using each brain region as a vertex on the graph, the connection strengths between a given vertex to all other vertices are compiled to form the signal vector for that vertex.*

the images are processed, they can be efficiently loaded using python libraries `nibabel` and `dipy`, and subsequently operated on using standard data analysis packages such as `numpy` and `scipy`.

Defining Graph Nodes and Features

Neuroimage data is readily applied to graph processing techniques and is often used as a benchmark application for new developments in graph processing [SNF+13]. Intuitively, the objective is to characterize the structural and functional relationships between brain regions, since correlations between PD and abnormal brain structure and function have been shown. As such, the first step is to define a graph structure for our data. This step alone has intuitive benefits. Even after preprocessing, individual voxels of MRI data contain significant noise that can affect analysis [GRS+16]. Brain region sizes vary greatly across individuals and change over one individual's lifetime (e.g., due to natural aging [Pet06]). Representing regions as vertices on a graph meaningfully groups individual voxels and mitigates these potential red herrings from analysis.

We use an undirected weighted graph $\mathcal{G} = \mathcal{V}, \mathcal{E}, \mathbf{W}$ with a set of vertices $\mathcal{V}$ with $|\mathcal{V}| =$ the number of brain regions N, a set of edges $\mathcal{E}$, and a weighted adjacency matrix $\mathbf{W}$, to represent our aMRI data. $\mathcal{G}$ is shared across the entire data set to represent general population-wide brain structure. Each vertex $v_i \in \mathcal{V}$ represents a brain region. Together, $\mathcal{V}, \mathcal{E}$, and $\mathbf{W}$ form a *k-Nearest Neighbor adjacency matrix*, in which each vertex is connected to its k nearest neighbors (including itself) by an *edge*, and edges are weighted according to the average Euclidean distance between two vertices. The weight values are normalized by dividing each distance by the maximum distance from a given vertex to all of its neighbors, $d_{ij} \in [0, 1]$. (Refer to Appendix B on our GitHub for details.)

`gen_nodes.py` first defines the vertices of the graph using the anatomical MRI data, which has been cleaned and *segmented* into brain regions by Freesurfer. The center voxel for each segmentation volume in each image is calculated. Next, `adj_mtx.py` calculates the mean center coordinate across all aMRI images for every brain region. The average center coordinate for each region i is a vertex $v_i \in \mathcal{V}$ of the graph $\mathcal{G}$. See Figure 1 for a depiction of the process.

Using these vertices, we wish to incorporate information from other modalities to characterize the relationships between the vertices. We define a *signal* on the vertices as a function $f : \mathcal{V} \to \mathbb{R}$, returning a vector $\mathbf{f} \in \mathbb{R}^N$. These vectors can be analyzed as "signals" on each vertex, where the change in signal across vertices is used to define patterns throughout the overall graph structure. In our case, the vector signal defined on a vertex v_i represents that vertex's weighted connectivity to all other vertices [SNF+13]. The weights correspond to the strength of connectivity between v_i and some other vertex v_j, as calculated by a given tractography algorithm. As such, each signal is a vertex of size N and there are N signals defined on each graph (one for each vertex), forming an N x N *weighted connectivity matrix*. Each dMRI image has one N x N set of signals for each tractography algorithm. In this way, the dimensionality of the data is drastically reduced, and information from multiple modalities and processing algorithms may be analyzed in a common data space.

`gen_features.py` approximates the strength of connectivity between each pair of vertices. For this, the number of tracts (output by each tractography algorithm) connecting each pair of brain regions must be counted. Recall that each image carries with it an affine matrix that translates the voxel data to a coordinate space. Each preprocessing software uses a different coordinate space, so a new affine matrix must be calculated to align the segmented anatomical images and the diffusion tracts (i.e., *coregistration*). Freesurfer's `mri_convert`, FSL's `flirt`, and DTK's `track_transform` are used to put the two modalities in the same coordinate space so that voxel-to-voxel comparisons can be made. Next, `nibabel`'s i/o functionality is used to generate a mask file for each brain region, `nibabel.streamlines` is used to read in the tractography data and `dipy.tracking.utils.target` is used to identify which tracts travel through each volume mask. The tracts are encoded using a unique hashing function to save space and allow later identification.

To generate the signals for each vertex, `utils.py` uses the encoded tract IDs assigned to each volume to count the number of tracts connecting each volume pair. The number of connections between pairs of brain regions approximate the connection strength, and these values are normalized similar to the normalization scheme mentioned above for the k-nearest neighbor weights. Figure 2 offers a visualization.

Graph Convolutional Networks

Common to many areas of data analysis, *spectral graph processing* techniques (i.e., processing a signal defined on a graph structure) have capitalized on the highly flexible and complex modeling

14. http://trackvis.org/dtk/

capacity of so-called deep learning neural network architectures. The layered construction of nonlinear calculations loosens rigid parameterizations of other classical methods. This is desirable, as changes in parameterizations have been shown to affect results in both neuroimage analysis (e.g., independent component analysis (ICA) [CJM+17]) and in graph processing (e.g., the explicit parameterization used in Chebyshev approximation [KW217]; further discussed in Appendix B).

In this paper, we utilize the Graph Convolutional Network (GCN) to compute signal processing on graphs. GCNs were originally used to classify the vertices of a single graph using a single set of signals defined on its vertices. Instead, our task is to learn signal patterns that generalize over many subjects' data. To this end, we designed a novel GCN architecture, which combines information from anatomical and diffusion MRI (dMRI) data, processes data from multiple diffusion MRI tractography algorithms for each dMRI image, and consolidates this information into a single vector so as to compare many subjects' data side-by-side. A single complete forward pass of our model consists of multiple parallel Graph Convolutional Networks (one for each tractography algorithm), max pooling, and graph classification via Graph Attention Network layers. We will briefly explain each part in this subsection; see Appendix B on our GitHub for a deeper discussion.

The convolution operation measures the amount of change enacted on a function f_1 by combining it with another function f_2. We can define f_2 such that its convolution with instances of f_1 from one class (e.g., PD) produce large changes while its convolution with instances of f_1 from another class (e.g., HC) produce small changes; this provides a way to discriminate instances of f_1 into classes without explicitly knowing the class values. Recall that we have defined a function f over the vertices of our graph using dMRI data (i.e., the *signals*). We seek to learn functions, termed *filters*, that, when convolved with the input graph signals, transform the inputs into distinguishable groups according to class value (e.g., PD vs. healthy control). This is similar to the local filters used in convolutional neural networks, except that the filters of GCNs use the connections of the graph (i.e., the edges) to establish locality.

Our specific implementation is based off the `GCN` class from [KW217]'s PyTorch implementation[15], which has several computational improvements over the original graph convolution formula. In short, the graph convolutional operation is based off the graph Laplacian

$$Ł = I - D^{\frac{-1}{2}}\mathbf{W}D^{\frac{-1}{2}},$$

where I is the identity matrix with 1's along the diagonal and 0's everywhere else, W is the weighted adjacency matrix defined earlier w.r.t. $\mathscr{G}$, and D is a weighted degree matrix such that $D_{ii} = \sum_j \mathbf{W}_{ij}$. We define the graph convolutional operation as

$$Z = \tilde{D}^{\frac{-1}{2}}\tilde{W}\tilde{D}^{\frac{-1}{2}}X\Theta,$$

A so-called *renormalization trick* has been applied to Ł wherein identity matrix I_N has been added; i.e., self-loops have been added to the adjacency matrix. $I_N + D^{\frac{-1}{2}}WD^{\frac{-1}{2}}$ becomes $\tilde{D}^{\frac{-1}{2}}\tilde{W}\tilde{D}^{\frac{-1}{2}}$, where $\tilde{W} = W + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{W}_{ij}$. $\Theta \in \mathbb{R}^{CxF}$ is a matrix of trainable coefficients, where $C = N$ is the length of the input signals at each node, and $F = N$ is the number of C-dimensional filters to be learned. X is the N x N matrix of input signals for all vertices (i.e., the signals from a single tractography output of a single dMRI image). $Z \in \mathbb{R}^{NxF}$ is the output matrix of convolved signals. We will call the output signals *features* going forward.

Generalizing Θ to the weight matrix $\mathbf{W}(l)$ at a layer, we can calculate a hidden layer of our GCN as

$$Z = f(X,A) = softmax(\hat{A}ReLU(\hat{A}X\mathbf{W}(0))\mathbf{W}(1)),$$

where $\hat{A} = \tilde{D}^{\frac{-1}{2}}\tilde{A}\tilde{D}^{\frac{-1}{2}}$.

Multi-View Pooling

For each dMRI acquisition, d different tractography algorithms are used to compute multiple "views" of the diffusion data. To account for the variability in the outputs produced by each algorithm, we wish to compile the information from each before classifying the whole graph. As such, d GCNs are trained side-by-side, such that the GCNs share their weights [[KZS15], [DMF+17]]. This results in d output graphs, i.e. d output vectors for each vertex. The vectors corresponding to the same vertex are pooled using max pooling, which has been shown to outperform mean pooling [ZHC18].

Graph Attention Networks

In order to convert the task from classifying each node to classifying the whole graph, the features on each vertex must be pooled to generate a single feature vector for each input. The *self-attention* mechanism, widely used to compute a concise representation of a signal sequence, has been used to effectively compute the importance of graph vertices in a neighborhood [VCC18]. This allows for a weighted sum of the vertices' features during pooling.

We employ a PyTorch implementation of [VCC18]'s `GAT` class[16] to implement a graph attention network, using a feed-forward neural network to learn attention coefficients as

$$a_{ij} = \frac{exp(LeakyReLU(a^T[\mathbf{W}_a h_i || \mathbf{W}_a h_j]))}{\sum_{k \in \mathscr{N}_i} exp(LeakyReLU(a^T[\mathbf{W}_a h_i || \mathbf{W}_a h_k]))},$$

where $||$ is concatenation.

Multi-Subject Training

The model is trained using `train.py`. First, several helper functions in `utils.py` are called to load the graph, input signals, and their labels, and prepare them for training. The model is built and run using the `GCNetwork` class in `GCN.py`. During training, the model reads in the signals for one dMRI acquisition at a time, where the signals from each tractography algorithm are processed in parallel, pooled into one graph, and then pooled into a single feature vector via the graph attention network. Using this final feature vector, a class prediction is made. Once a class prediction is made for every input dMRI instance, the error is computed and the weights of the model are updated through backpropagation. This is repeated over many epochs to iteratively fit the weights to the classification task. Figure 3 shows an outline of the network architecture.

15. https://github.com/tkipf/pygcn

16. https://github.com/Diego999/pyGAT

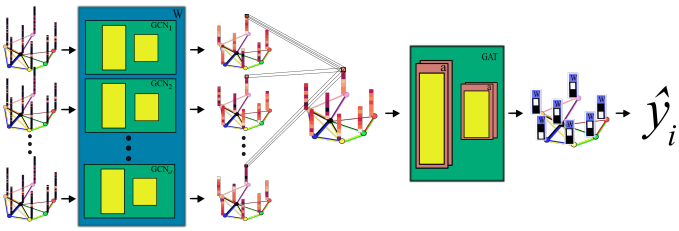


Fig. 3: *A depiction of the novel GCN architecture is shown. First, a GCN is trained for each "view" of the data, corresponding to a specific tractography algorithm. The GCNs share weights, and the resulting features are pooled for each vertex. This composite graph is then used to train a multi-head graph attention network, which assigns a weight (i.e., "attention") to the feature computed at each vertex. The weight assigned to each vertex is used to compute a weighted sum of the features, yielding a single feature vector for graph classification.*

Methods

Our data is downloaded from the Parkinson's Progression Markers Initiative (PPMI)[17] database. We download 243 images, consisting of 96 aMRI images and 140 diffusion images. The images are from 20 individuals (each subject had multiple visits to the clinic and data from multiple image modalities). Among the images, 117 are from the Parkinson's Disease (PD) group and 30 are from healthy controls (HC). We preprocessed our data using the pipeline described above. We ran this preprocessing using a Google cloud virtual machine with 96 CPU cores over the course of several days.

Following preprocessing, we constructed the shared adjacency matrix and trained the model on the dMRI signals, which totaled to 588 (147 dMRI acquisitions x 4 tractography algorithms) N x N connectivity matrices. We calculated the adjacency matrix using each node's 20 nearest neighbors. To account for the class imbalance between PD and HC images, we use a bagging method. On each of five iterations, all the images from the HC group were combined with an equally-sized subset from the PD group. All of the images were used at least once during training, and the overall performance measures were averaged across training folds.

Using caution to prevent any forms of data leakage, we used a roughly 80/20 train-test split, wherein we ensured all data from the same subject was used as only training or testing data. To assess the performance of our GCN model, we first trained a number of baseline models on the features constructed from the diffusion data. These models include k-nearest neighbor, logistic regression, ridge regression, random forest, and support vector machine (SVM, with both linear and polynomial kernels) from `scikit-learn`; we also trained a fully-connected neural network (fcNN) and a 4-channel convolutional neural network (CNN) using `PyTorch`. Finally, we compare our model to the "siamese multi-view" GCN (sMVGCN) used in [ZHC18]. This network utilizes diffusion and anatomical MRI data and trains on pairs of image data to predict whether the pairs are from the same or different classes. The data is also from the PPMI data set and uses the PD and HC classes during classification. This was the closest model to ours that we found in the literature.

Except for the multi-channel CNN, we trained each model on the features from each tractography algorithm individually, and averaged the results. We calculated the overall accuracy, F1 score, and area under the ROC curve (AUC) as our performance measures. The default parameters were used for the `scikit-learn` models. The fcNN was a three-layer network with two hidden layers. The first layer had 128 ReLU units; the second had 64. For the CNN, a single convolutional layer was used, containing 18 filters of size 3; stride of 1 was used. Max pooling with a kernel size of 2 and stride of 2 was used to feed the features through two fully-connected layers before the final output. The first fully-connected layer reduced the 18x57x57-dimension input - where 57 is the original input width and height of 115 halved via max pooling - to 64 ReLU hidden units. For both neural networks, softmax activation was applied to the outputs and negative log likelihood was used as the loss function (i.e., cross entropy). Again for both models, learning rate was set to 0.01 and dropout of 0.5 was used between fully-connected hidden layers. These parameters coincide with the default parameters of the graph convolutional network class we used, and are commonly used in the literature. We used a validation set to find the optimal number of epochs to train each network for. We tested 40, 80, 100, 140, 200, and 400 epochs for each model and found that 140 worked best for the fcNN, and 100 for the CNN.

We trained the graph convolutional network (GCN) on the same bagged subsets of data for comparison purposes. The only difference is that the features are md to the vertices of the adjacency matrix before training. We used a validation set to tune the model parameters. We tested with or without dropout (set to 0.5 when used), with or without weight decay (set to 5e-4 when used), the number of hidden units for the first GCN layer (8,16,32), the number of "heads" or individual attention weights (2,4,6,8), and the number of epochs (same options as for the fcNN and GCN). We found that dropout of 0.5, weight decay of 5e-4, 8 hidden units, 8 attention heads, and 80 epochs worked best for our model. The results from training the GCN are also included in Table 1.

Results

The results from training the diffusion data on baseline models, and the combined diffusion and anatomical data on the GCN are included in Table 1. We report accuracy, F1-score, and AUC for each model; these numbers are averaged across five training iterations using subsets of the data to account for class imbalance. Subsequently, we analyze the attention weights from the GCN model. Each node of the adjacency matrix was assigned an attention weighting corresponding to that node's importance in determining the overall class of the graph. Since each node of the adjacency matrix corresponds to an anatomical brain region, we could interpret the magnitude of each node's attention weight as the relative importance of a brain region for distinguishing the PD vs. HC classes. We compiled the attention weights from each training iteration and determined the 16 brain regions with the highest weights. The names and relative importance assigned to these regions are shown in Figure 4.

Discussion and Conclusion

From the results on the baseline models, we can see that the features generated from the diffusion MRI data are suitable for distinguishing the PD vs. HC classes. For example, the relatively high performance of the SVM models demonstrate that the features are roughly linearly separable. Furthermore, we see from the improved performance of the GCN model that the incorporation of anatomical data improves the capacity for the data to be modeled. Of the 16 highest-weighted regions according to the

17. https://www.ppmi-info.org

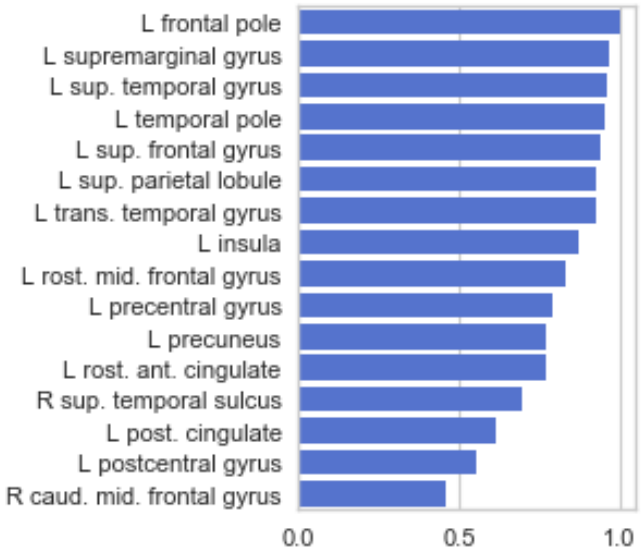


Fig. 4: *The 16 regions with highest attention weighting across all training iterations are shown. "L" and "R" indicate regions on the left or right hemisphere, respectively. "post.", "ant.", "sup.", "mid.", "rost.", "caud.", and "trans." indicate posterior, anterior, superior, middle, rostral, caudal, and transverse, respectively.*

Model	Accuracy (%)	F1-Score	AUC
k-Nearest Neighbor	63.66%	0.636	0.646
Logistic Regression	75.72%	0.749	0.839
Ridge Regression	85.54%	0.883	0.500
Random Forest	77.77%	0.765	0.782
SVM (linear kernel)	87.66%	0.873	0.894
SVM (polynomial kernel)	87.02%	0.899	0.887
Fully-Connected NN	83.98%	0.854	0.881
Convolutional NN	85.33%	0.900	0.908
Graph Convolutional NN	**92.14%**	**0.953**	**0.943**

TABLE 1: *The results from our testing of the baseline algorithms on the features constructed from the diffusion data alone, and our graph convolutional network (GCN) which additionally incorporates anatomical information. The results are averaged across five training iterations, which use subsamples of the data to ensure class balance.*

GAT attentions layers, 9 coincide with lateral or contralateral regions identified by [ZHC18] as significantly contributing to the distinction between PD and HC classes. All but two of the regions listed in Figure 4 were from the left hemisphere, whereas the majority of regions in [ZHC18] were from the right hemisphere. We aren't sure why this may be, but the stronger identification of left hemispheric regions aligns with asymmetries found by [CMD+16], wherein the left hemisphere is more significantly affected in early-stage PD.

Due to the time required to construct the pipeline, and the substantial time and compute resources required for each additional image, we used a relatively small data set. The models showed signs of overfitting during training, due to increasing performance on the training data after improvement with the testing data had stopped. We feel that reproduction with a larger dataset may mitigate this issue and improve the robustness of our initial results.

We would also like to see future studies incorporate both diffusion and functional MRI data. We investigated the use of the C-PAC preprocessing software to generate features from functional MRI (fMRI) data, and we believe these features could be incorporated into our model. Additional anatomical information such as the volume of each region could also be incorporated, and even metadata such as age or genetic information could be added to each node of an image to encourage class separation. These points reflect our use of graph convolutional networks for multimodal neuroimage analysis, as the format allows for the combination of multiple forms of data in an efficient and intuitive manner. All of these points were beyond the scope of the current experiment, and are possibilities for future research.

We have made the code for our pipeline available on GitHub. Included in the repository are the parameters we used to download our data from PPMI, so that researchers with access to the database might download similar data for reproduction. Processing this data is very technical; there are multiple ways of doing so and our pipeline is surely capable of being improved upon. For example, we utilized all 115 brain regions returned by Freesurfer's segmentaion. Instead, [ZHC18] selectively utilize only 84 of the regions. By confining the number of regions, e.g., to only those with clinical significance to PD, we may see improvements in performance and interpretability.

We have presented here a complete pipeline for preprocessing multi-modal neuroimage data, applied to real-world data aimed at developing image biomarkers for Parkinson's disease research. We propose a novel graph-based deep learning model for analysing the data in an interpretable format. Our focus in this paper was to explicitly delineate the steps we took and implement sound data analysis techniques, so as to enable reproducibility in the field. To this end, we hope to help bridge the gap between neuroscience research and advanced data analysis.

Acknowledgements

Data used in the preparation of this article were obtained from the Parkinson's Progression Markers Initiative (PPMI) database (www.ppmi-info.org/data). For up-to-date information on the study, visit www.ppmi-info.org. PPMI - a public-private partnership - is funded by the Michael J. Fox Foundation for Parkinson's Research and funding partners, including Abbvie, Allergan, Avid, Biogen, BioLegend, Bristol-Mayers Squibb, Colgene, Denali, GE Healthcare, Genentech, GlaxoSmithKline, Lilly, Lundbeck, Merck, Meso Scale Discovery, Pfizer, Piramal, Prevail, Roche, Sanofi Genzyme, Servier, Takeda, TEVA, UCB, Verily, Voyager, and Golub Capital.

References

[BP14] J Blesa and S Przedborski. Parkinson's disease: animal models and dopaminergic cell vulnerability. *Frontiers Neuroanatomy*, 8, 2014. doi:10.3389/fnana.2014.00155.

[BPP+00] PJ Basser, S Pajevic, C Pierpaoli, J Duda, and A Aldroubi. In vivo fiber tractography using dt-mri data. *Magn. Reson. Med.*, 44:625–632, 2000.

[Bro16] Per Brodal. *The Central Nervous System, Fifth edition*. Oxford University Press, 2016.

[BSS+18] RG Burciu, RD Seidler, P Shukla, MA Nalls, AB Singleton, MS Okun, and DE Vaillancourt. Multimodal neuroimaging and behavioral assessment of a-synuclein polymorphism rs356219 in older adults. *Neurobiol Aging*, 66:32–39, 2018. doi:10.1016/j.neurobiolaging.2018.02.001.

[BWS+11] S Baudrexel, T Witte, C Seifried, F von Wegner, F Beissner, JC Klein, H Steinmetz, R Deichmann, J Roeper, and R Hilker. Resting state fmri reveals increased subthalamic nucleus-motor cortex connectivity in parkinson's disease. *Neuroimage*, 55(4):1728–1738, 2011. doi:10.1016/j.neuroimage.2011.01.017.

[CJM+17] M Cousineau, PM Jodoin, FC Morency, V Rozanski, M Grand'Maison, BJ Bedell, and M Descoteaux. A test-retest study on parkinson's ppmi dataset yields statistically significant white matter fascicles. *NeuroImage: Clinical*, 16, 2017. doi:10.1016/j.nicl.2017.07.020.

[CLC99] *Tracking neuronal fiber pathways in the living human brain*, volume 96. Proceedings of the National Academy of Sciences of the United States of America, 1999. doi:10.1073/pnas.96.18.10422.

[CMD+16] Daniel O Claassen, Katherine E McDonell, Rawal Donahue, Neimat Wylie, Hakmook Kang, Peter Hedera, David Zald, Bennett Landman, Benoit Dawant, and Swati Rane. Cortical asymmetry in parkinson's disease: early susceptibility of the left hemisphere. 6(12), 2016. doi:10.1002/brb3.573.

[DDH16] F DuBois Bowman, Daniel F Drake, and Daniel E Huddleston. Multimodal imaging signatures of parkinson's disease. *Frontiers Neuroscience*, 10, 2016. doi:10.3389/fnins.2016.00131.

[DMF+17] M Descoteaux, L Maier-Hein, A Franz, P Jannin, D Collins, and S Duchesne, editors. *Distance Metric Learning using Graph Convolutional Networks: Application to Functional Brain Networks*, volume 10433 of *Lecture Notes in Computer Science*. Medical Image Computing and Computer Assisted Intervention (MICCAI 2017), Springer, Cham, 2017.

[GGL+18] K Gmitterova, J Gawinecka, F Llorens, D Varges, P Valkovic, and I Zerr. Cerebrospinal fluid markers analysis in the differential diagnosis of dementia with lewy bodies and parkinson's disease dementia. *European Archives of Psychiatry and Clinical Neuroscience*, 2018. doi:10.1007/s00406-018-0928-9.

[GLH+14] R Geevarghese, DE Lumsden, N Hulse, M Samuel, and K Ashkan. Subcortical structure volumes and correlation to clinical variables in parkinson's disease. *Journal of Neuroimaging*, 25(2), 2014. doi:10.1111/jon.12095.

[GRS+16] L Griffanti, M Rolinski, K Szewczyk-Krolikowski, RA Menke, N Filippini, G Zamboni, M Jenkinson, MTM Hu, and CE Mackay. Challenges in the reproducibility of clinical studies with resting state fmri: An example in early parkinson's disease. *Neuroimage*, 124(Pt A):704–703, 2016. doi:10.1016/j.neuroimage.2015.09.021.

[HBL10] Ray H Hashemi, William G Bradley Jr., and Christopher J Lisanti. *MRI: The Basics, Third edition*. Lippincott Williams & Wilkins, 2010.

[HCS+14] RD Hjelm, V Calhoun, R Salakhutdinov, E Allen, T Adali, and S Plis. Restricted boltzmann machines for neuroimaging: An application in identifying intrinsic networks. *NeuroImage*, 96, 2014. doi:10.1016/j.neuroimage.2014.03.048.

[HDC+18] RD Hjelm, E Damaraju, K Cho, H Laufs, S Plis, and V Calhoun. Spatio-temporal dynamics of intrinsic networks in functional magnetic imaging data using recurrent neural networks. *Frontiers in Neuroscience*, 12(600), 2018. doi:10.3389/fnins.2018.00600.

[HKK90] Eric J. Hartman, James D. Keeler, and Jacek M. Kowalski. Layered neural networks with gaussian hidden units as universal approximations. *Neural Computation*, 2(2):210–215, 1990. doi:10.1162/neco.1990.2.2.210.

[KCR16] Vijay Kumar BG, Gustavo Carneiro, and Ian Reid. Learning local image descriptors with deep siamese and triplet convolutional networks by minimizing global loss functions. *IEEE CVPR*, page 5385–5394, 2016.

[KPF+18] Sofia Ira Ktena, Sarah Parisot, Enzo Ferrante, Martin Rajchl, Matthew Lee, Ben Glocker, and Daniel Rueckert. Metric learning with spectral graph convolutions on brain connectivity networks. *NeuroImage*, 169:431–442, 2018. doi:10.1016/j.neuroimage.2017.12.052.

[KUH+16] J Kleesiek, G Urban, A Hubert, D Schwarz, K Maier-Hein, M Bendszus, and A Biller. Deep mri brain extraction: A 3d convolutional neural network for skull stripping. *NeuroImage*, 129, 2016. doi:10.1016/j.neuroimage.2016.01.024.

[Kum18] K Kumar. *Data driven methods for characterizing individual differences in brain MRI*. PhD thesis, 2018. doi:10.13140/RG.2.2.12193.71525.

[KW217] *Semi-Supervised Classification with Graph Convolutional Networks*. ICLR, 2017.

[KZS15] *Siamese Neural Networks for One-shot Image Recognition*, volume 37, Lille, France, 2015. JMLR: W&CP.

[LCL+15] Sidong Liu, Weidong Cai, Siqi Liu, Fan Zhang, Michael Fulham, Dagan Feng, Sonia Pujol, and Ron Kikinis. Multimodal neuroimaging computing: a review of the applications in neuropsychiatric disorders. *Brain Inform*, 2(3):167–180, 2015. doi:10.1007/s40708-015-0019-x.

[LMS+18] Juliette H Lanskey, Peter McColgan, Anette E Schrag, Julio Acosta-Cabronero, Geraint Rees, Huw R Morris, and Rimona S Weil. Can neuroimaging predict dementia in parkinson's disease? *Brain*, 141(9), 2018. doi:doi.org/10.1093/brain/awy211.

[LSC+14] C Luo, W Song, Q Chen, Z Zheng, K Chen, B Cao, J Yang, J Li, X Huang, Q Gong, and HF Shang. Reduced functional connectivity in early-stage drug-naive parkinson's disease: a resting-state fmri study. *Neurobiol Aging*, 35(2):431–441, 2014. doi:10.1016/j.neurobiolaging.2013.08.018.

[LWT+03] M Lazar, DM Weinstein, JS Tsuruda, KM Hasan, K Arfanakis, ME Meyerand, B Badie, HA Rowley, V Haughton, A Field, and AL Alexander. White matter tractography using diffusion tensor deflection. *Hum. Brain Mapp.*, 18:306–321, 2003. doi:10.1002/hbm.10102.

[LWX+12] Dan Long, Jinwei Wang, Min Xuan, Quanquan Gu, Xiaojun Xu, Dexing Kong, and Minming Zhang. Automatic classification of early parkinson's disease with multi-modal mr imaging. *PLOS ONE*, 7(11), 2012. doi:doi.org/10.1371/journal.pone.0047714.

[LZC17] Deep learning and convolutional neural networks for medical image computing, 2017.

[MCCv99] S Mori, BJ Crain, VP Chacko, and PC van Zijl. Three-dimensional tracking of axonal projections in the brain by magnetic resonance imaging. *Ann. Neurol.*, 45:265–269, 1999.

[MLL+18] DC Matthews, H Lerman, A Lukic, RD Andres, A Mirelman, MN Wenick, N Giladi, SC Strother, KC Evans, JM Cedarbaum, and E Even-Sapir. Fdg pet parkinson's disease-related pattern as a biomarker for clinical trials in early stage disease. *NeuroImage: Clinical*, 2018. doi:10.1016/j.nicl.2018.08.006.

[Pet06] R Peters. Ageing and the brain. *Postgraduate Medical Journal*, 82(964):84–88, 2006. doi:10.1136/pgmj.2005.036665.

[RST14] Amy Reeve, Eve Simcox, and Doug Turnbull. Ageing and parkinson's disease: Why is advancing age the biggest risk factor? *Elsevier Sponsored Documents Ageing Research Reviews*, 14(100):19–30, 2014. doi:10.1016/j.arr.2014.01.004.

[SNF+13] D I Shuman, S K Narang, P Frossard, A Ortega, and P Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013. doi:10.1109/MSP.2012.2235192.

[TBvE+15] M Tahmasian, LM Bettray, T van Eimeren, A Drzezga, L Timmermann, CR Eickhoff, SB Eickhoff, and C Eggers. A systematic review on the applications of resting-state fmri in parkinson's disease: Does dopamine replacement therapy play a role? *Cortex*, 73:80–105, 2015. doi:10.1016/j.cortex.2015.08.005.

[VCC18] *Graph Attention Networks*. ICLR, 2018.

[Wan15] Yi Wang. *Principles of Magnetic Resonance Imaging: Physics concepts, pulse sequences and biomedical applications*. 2015.

[ZHC18] *Multi-View Graph Convolutional Network and Its Applications on Neuroimage Analysis for Parkinson's Disease*. AMIA Annual Symposium Proceedings Archive, 2018.

[ZYH+18] M Zhong, W Yang, B Huang, W Jiang, X Zhang, X Liu, L Wang, J Wang, L Zhao, Y Zhang, Y Liu, J Lin, and R Huang. Effects of levodopa therapy on voxel-based degree centrality in parkinson's disease. *Brain Imaging and Behavior*, 2018. doi:10.1007/s11682-018-9936-7.

[ZZW+15] Liang Zhan, Jiayu Zhou, Yalin Wang, Yan Jin, Neda Jahanshad, Gautam Prasad, Talia M Nir, Cassandra D Leonardo, Jieping Ye, and Paul M Thompson. Comparison of nine tractography algorithms for detecting abnormal structural brain networks in alzheimer's disease. *Frontiers in Aging Neuroscience*, 7(48), 2015. doi:10.3389/fnagi.2015.00048.

pyjanitor: A Cleaner API for Cleaning Data

Eric J. Ma[‡*], Zachary Barry[‡], Sam Zuckerman, Zachary Sailer[§]

Abstract—The `pandas` library has become the de facto library for data wrangling in the Python programming language. However, inconsistencies in the `pandas` application programming interface (API), while idiomatic due to historical use, prevent use of expressive, fluent programming idioms that enable self-documenting `pandas` code. Here, we introduce `pyjanitor`, an open source Python package that extends the `pandas` API with such idioms. We describe its design and implementation of the package, provide usage examples from a variety of domains, and discuss the ways that the `pyjanitor` project has enabled the inclusion of first-time contributors to open source projects.

Index Terms—data engineering, data science, data cleaning

Introduction

Data preprocessing, or data wrangling, is an unavoidable task in data science. It is a common experience amongst data scientists that data wrangling can occupy up to 80% of their time [nyt] [Wic14]. Part of this time is spent defining modelling approaches, and part of this time is writing code that executes the sequence of transformations on raw data that wrangle it into the necessary shape for downstream modelling work.

In the Python ecosystem, `pandas` is the *de facto* tool for data manipulation. This is because it provided an API for manipulating tabular data when conducting data analysis. This API was noticeably missing from the Python standard library and NumPy, which, prior to `pandas` emergence, were the primary tools for data analysis in Python. Hence, through the `DataFrame` object and its interfaces, `pandas` provided a key API that enabled statisticians, data scientists, and machine learners to wrangle their data into a usable shape. That said, there are inconsistencies in the *pandas* API which, though now are idiomatic due to historical use, prevent the use of expressive, fluent [flu] programming idioms[1] that enable self-documenting data science code.

Idiomatic Inconsistencies of `pandas`

A case in point is the following elementary sequence of data preprocessing operations:

1) Standardizing column names to snake-case (`spelled_like_this`, rather than `Spelled! Like! This?`),
2) Removing unnecessary columns,
3) Adding a column of data,
4) Log-transforming a column,
5) Filtering the log-transformed column,
6) Dropping rows that have null values,
7) Adding a column that is the mean of each sample's group.

To do this with the pandas API, one might write the following code.

```
import pandas as pd
import numpy as np
import re

df = pd.DataFrame(...)

def clean_name(x):
    """Custom function to sanitize column name."""
    FIXES = [(r"[\* /:,?!()\.-]", "_"), (r"['']", "")]
    for search, replace in FIXES:
        x = re.sub(search, replace, x)
    return x.lower().replace('__', '_')

df = (
    df
    # clean column names
    .rename(columns=clean_name)
    # remove column
    .drop('column_name_14', axis='columns')
    # log transform
    .assign(
        column_name_13=lambda x: np.log10(x['column_name_13'])
    )
    # drop null values
    .dropna()
    # filter based on column value
    .query("column_name_13 < 3")
)

# add a column that is the mean of each sample's group.
col13_means = df.groupby('group').mean()['column_name_13']
df = df.join(col13_means, rsuffix='_mean', on='group')
```

By using `pyjanitor`, end-users can instead write code that reads much closer to the plain English description.

```
import pandas as pd
import numpy as np
import janitor

df = (
    pd.DataFrame(...)
    .clean_names()
    .remove_column('column_name_14')
    .transform_column('column_name_13', np.log10)
    .query('column_name_13 < 3')
    .dropna()
    .groupby_agg(
        by="group",
        agg_column_name="column_name_13",
```

* Corresponding author: ericmajinglong@gmail.com
‡ Novartis Institutes for Biomedical Research
§ Jupyter Project

1. Fluent interfaces, as a term, were first coined in 2006, and describe a programming pattern allowing code to more closely resemble written prose. Method chaining is the most common way to achieve this.

```
        new_column_name="column_name_13_mean",
        agg="mean",
    )
)
```

This is the API design that `pyjanitor` aims to provide to `pandas` users: common data cleaning routines that can be mix-and-matched with existing `pandas` API calls. This is in keeping with Line 7 of the Zen of Python, which states that "Readability counts"; `pyjanitor` thus enables data scientists to construct their data processing code with an easily-readable sequence of meaningful verbs. By providing commonly-usable data processing routines, we also save time for data scientists and engineers, allowing them to accomplish their work more efficiently.

History of `pyjanitor`

`pyjanitor` started as a Python port of the R package `janitor`, which provides the same functionality to R users. The initial goal was to explicitly copy the `janitor` function names while engineering it to be compatible with `pandas.DataFrames`, following Pythonic idioms, such as the method chaining provided by some `pandas` class methods. As the project evolved, the scope broadened, to provide a defined language for data processing as an extension on `pandas` DataFrames, including submodules with functions specific for bioinformatics, cheminformatics, and finance.

Architecture

`pyjanitor` relies completely on the `pandas` extension API (https://pandas.pydata.org/pandas-docs/stable/development/extending.html), which allows developers to create functions that behave as if they were native `pandas.DataFrame` class methods. The only requirement here for such functions is that the first argument to it be a `pandas.DataFrame` object:

```
def data_cleaning_function(df, **kwargs):
    ...
    # data cleaning functions go here
    ...
    return df
```

In order to reduce the amount of boilerplate required, `pyjanitor` also makes heavy use of `pandas_flavor` [pf], which provides an easy-to-use function decorator that handles class method registration. As such, to extend the `pandas` API with more instance-method-like functions, we only have to decorate the custom function, as illustrated in the following code sample:

```
import pandas_flavor as pf

@pf.register_dataframe_method
def data_cleaning_function(df, **kwargs):
    ...
    # data cleaning operations go here
    ...
    return df
```

`pandas-flavor` has functionality that warns, at runtime, whether a `DataFrame` attribute has been overwritten by a custom function. Our test suite allows us to catch this issue before committing contributed code to the library.

Underneath each data cleaning function, we are free to use both the imperative and functional APIs. What is exposed, then, is a functional and fluent API for the end-user.

Thanks to the `pandas.DataFrame.query()` API, symbolic evaluations are generally available in `pyjanitor` for filtering data. This enables us to write functions that do filtering of the DataFrame using a verb that might match end-users' intuitions better. One such example is the `.filter_on('criteria')` method, illustrated in the opening example.

Design

Inspired by the `dplyr` world, `pyjanitor` functions are named with verb expressions. This, as mentioned earlier, this helps with readability. Hence, if we want to "clean names", the end user can call on the `.clean_names()` function/class method. If the end user wants to "remove all empty rows and columns", they can call on `.remove_empty()`. As far as possible, function names are expressed using simple English verbs that are understandable cross-culturally and well-documented, to ensure that this API is inclusive and accessible to the widest subset of users possible.

Where domain-specific verbs are used, we strive to match the mental models and vocabulary of domain experts. One example comes from the `biology` submodule, where the `join_fasta` function allows a bioinformatics-oriented user to add in a column of sequences based on FASTA accession numbers that are keys for sequence values in a FASTA-formatted file [PL88].

Keyword arguments are also likewise named with verb expressions where relevant. For example, if one wants to preserve and record the original column names before cleaning, one can add the `preserve_original` keyword argument to the `.clean_names` method:

```
(
    df
    .clean_names(
        preserve_original=True,
        remove_special=True,
        ...
    )
)
```

In order to adhere to a functional programming paradigm, no operations that change the original DataFrame are allowed. Hence, if the internal implementation of a function results in a mutation of the original DataFrame, we explicitly make a copy of the DataFrame first, though we also generally try to avoid double-copying as well. This decision, which was made after a fairly extensive discussion on our issue tracker, balances functional design principles and pragmatic considerations when dealing with potentially large dataframe objects.

A final design choice we made was to explicitly disallow overriding or duplicating existing DataFrame class methods. The goal here is to extend `pandas`, rather than replace its API, and we have turned down user requests to do so.

Documentation

Full API Documentation for *pyjanitor* is available on ReadTheDocs [doc].

An examples gallery, which contains Jupyter notebooks that showcase how to use `pyjanitor`, is also part of the documentation.

Development

The reception to `pyjanitor` has been encouraging thus far. Newcomer contributors to open source have made their first contributions to `pyjanitor`, and experienced software developers have also chipped in. Many contributors are data scientists

themselves, who are also seeking cleaner APIs to help them get their work done. There is a salient lesson here: with open source tools, savvy users can help steer development in a direction that they need, and we would encourage other contributors to join in too.

As with most open source software development, maintenance and new feature development are entirely volunteer driven. Users are invited to post feature requests on the source repository issue tracker, but are more so invited to contribute an implementation themselves to share. To date, 31 contributors have made pull requests into `pyjanitor`, and we look forward to further contributions being made at the SciPy conference sprints.

In the spirit of being beginner-friendly, new contributions to the pyjanitor library are encouraged to solve one and only one specific problem first, before we figure out how to either (1) generalize the function use case, or (2) generalize the implementation.

As an example, the commit history for `clean_names()` follows this pattern. The initial implementation manually listed out every character to be replaced by an underscore, in a DataFrame with a single column level. A later pull request extended the implementation to multi-level columns, and the current improved version uses regex string replacement to concisely express the cleaning operation. Most notably, each of these contributions were made by first-time open source contributors.

For the long-term health of the package, we are on the lookout for underrepresented contributors who would like to help maintain the package long-term as well. A code of conduct document, and a community guidelines document, are also on our development roadmap.

Other Related Tools

When developing `pyjanitor`, we originally set out to port `janitor` (the R package) to Python, providing compatibility with `pandas` DataFrames in a style compatible with Pythonic idioms (e.g. method chaining). While development was under way, we also found the Python alternatives described below, and found them to either (a) be lacking active development, (b) inventing a new pipe-like operator, (c) be restricted to dplyr verbs, and/or (d) lacking a robust community of developers. Hence, the development of `pyjanitor` was, and still is, oriented towards solving these problems.

For the convenience of our readers, we list our assessment of related tools below.

janitor [jan]: This is the original source of inspiration for `pyjanitor`, and the original creator of `janitor` is aware of `pyjanitor`'s existence. A number of function names in `janitor` have been directly copied over to `pyjanitor` and re-implemented in a `pandas`-compatible syntax.

dplyr [dplb]: The `dplyr` R package can be considered as "the originator" for verb-based data processing syntax. `janitor` the R package extends `dplyr`. It is available for use by Python users through `rpy2`; however, its primary usage audience is R users.

pandas-ply [pan]: This is a tool developed by Coursera, and aims to provide the `dplyr` syntax to `pandas` users. One advantage that it has over `pyjanitor` is that symbolic expressions can be used inside functions, which automatically get parsed into an appropriate lambda function in Python. However, it is restricted to the `dplyr` verb set.

dplython [dpla]: Analogous to `pandas-ply`, `dplython` also aims to provide the `dplyr` syntax to *pandas* users, but just like `pandas-ply`, it is restricted to `dplyr` verbs.

dfply [dfp]: This is the most actively-developed, pandas-compatible `dplyr` port. Its emphasis is on porting over the piping syntax to the pandas world. From our study of its source code, in principle, every function there can be wrapped with `pandas-flavor`'s `.register_dataframe_method` decorator, thus bringing the most feature-complete implementation of `dplyr` verbs to the `pandas` world. It does, however, re-implement a number of `pandas` functions using `dplyr` names. This makes it distinct from the pyjanitor project, where extension, rather than replacement, of existing `pandas` functionality is generally encouraged. Whether the developers are interested in collaboration remains to be discussed.

plydata [ply]: Like the others mentioned before, `plydata` also aims to provide the `dplyr`-style data manipulation grammar to `pandas`. It also provides a *pipe*-like operator (`>>`), and features integration with `plotnine`, a grammar of graphics plotting library for the Python programming language.

kadro [kad]: Kadro uses a wrapper around `pandas.DataFrame` objects to provide `dplyr`-style syntax.

pdpipe [pdp]: `pdpipe` provides a language for creating data preprocessing pipelines that are turned into Python callables, through which a DataFrame can be passed. Its design choice is to create fluent pipelines as pre-declared functions that are chained, rather than as methods that are attached onto a DataFrame. This distinction separates `pyjanitor` and `pdpipe`.

Limitations of `pyjanitor`

A current technical limitation of `pyjanitor` is the inability to symbolically parse expression strings to perform column-wise transformations. An example of a desired API might be:

```
df = (
    pd.DataFrame(...)
    .mutate(
        expression="column_name_12 + column_name_13",
        new_column_name="summation"
    )
)
```

As of now, because symbolic parsing is unavailable, this fluent and declarative syntax that is available to `dplyr` users is unavailable to `pyjanitor` users. We would welcome a contribution that enables this, perhaps using the `patsy` package.

Extensions beyond `pyjanitor`

`pyjanitor` does not aim to be the all-purpose data cleaning tool for all subject domains. Apart from providing a library of generally useful data manipulation and cleaning routines, one can also think of the project as a catalyst project for other specific domain applications. Following the verb-based grammar, one may imagine even more specific domain terms. Hence we have developed domain-specific submodules with a view towards encouraging their further development as independent packages.

For example, in our `chemistry` submodule, we have the following functions implemented that aid in cheminformatics-oriented data science tasks:

- `smiles2mol(df, col_name)`: to convert a column of smiles into RDKit [rdk] mol objects.
- `mol2graph(df, col_name)`: to convert a column of mol objects into NetworkX [HSS08] graph objects.

In our biology submodule, convenience functions exist to accomplish the following tasks:

- `join_fasta(df, file_name, id_col, col_name)`: create a column that contains the string representation of a biological sequence, by "joining" in a FASTA file, mapping the string to a particular column that already has the sequence identifiers in it.

The dependencies required for their usage are optional at install-time, and we provide instructions for end-users to install the relevant packages if they are not already installed locally.

Acknowledgments

We would like to thank the users who have made contributions to `pyjanitor`. These contributions have included documentation enhancements, bug fixes, development of tests, new functions, and new keyword arguments for functions. The list of contributors, which we anticipate will grow over time, can be found under `AUTHORS.rst` in the development repository.

We would also like to acknowledge the tremendous convenience provided by `pandas-flavor`, which was developed by one of our co-authors, Dr. Zachary Sailer.

REFERENCES

[dfp] dplyr-style piping operations for pandas dataframes. https://github.com/kieferk/dfply. Accessed: 24 November 2019.

[doc] pyjanitor documentation. https://pyjanitor.readthedocs.io. Accessed: 22 May 2019.

[dpla] Dplython: Dplyr for python. https://github.com/dodger487/dplython. Accessed: 22 May 2019.

[dplb] A grammar of data manipulation: dplyr. https://dplyr.tidyverse.org. Accessed: 22 May 2019.

[flu] Fluentinterface. https://martinfowler.com/bliki/FluentInterface.html. Accessed: 22 May 2019.

[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.

[jan] janitor: Simple tools for examining and cleaning dirty data. https://github.com/sfirke/janitor. Accessed: 22 May 2019.

[kad] A friendly pandas wrapper with a more composable grammar support. https://github.com/koaning/kadro. Accessed: 22 May 2019.

[nyt] For big-data scientists, 'janitor work' is key hurdle to insights. https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html. Accessed: 22 May 2019.

[pan] pandas-ply: functional data manipulation for pandas. https://github.com/coursera/pandas-ply. Accessed: 24 November 2019.

[pdp] https://github.com/shaypal5/pdpipe. Accessed: 22 May 2019.

[pf] Pandas flavor: The easy way to write your own flavor of pandas. https://github.com/Zsailer/pandas_flavor. Accessed: 22 May 2019.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. U.S.A.*, 85(8):2444–2448, Apr 1988.

[ply] A grammar for data manipulation in python. https://github.com/has2k1/plydata. Accessed: 22 May 2019.

[rdk] Rdkit: Open-source cheminformatics. http://www.rdkit.org. Accessed: 22 May 2019.

[Wic14] Hadley Wickham. Tidy data. *Journal of Statistical Software, Articles*, 59(10):1–23, 2014. URL: https://www.jstatsoft.org/v059/i10, doi:10.18637/jss.v059.i10.

Codebraid: Live Code in Pandoc Markdown

Geoffrey M. Poore[‡*]

Abstract—Codebraid executes code blocks and inline code in Pandoc Markdown documents as part of the document build process. Code can be executed with a built-in system or Jupyter kernels. Either way, a single document can involve multiple programming languages, as well as multiple independent sessions or processes per language. Because Codebraid only uses standard Pandoc Markdown syntax, Pandoc handles all Markdown parsing and format conversions. In the final output document produced by Pandoc, a code chunk can be replaced by a display of any combination of its original Markdown source, its code, the stdout or stderr resulting from execution, or rich output in the case of Jupyter kernels. There is also support for programmatically copying code or output to other parts of a document.

Index Terms—reproducibility, dynamic report generation, literate programming, Python, Pandoc, Project Jupyter

Introduction

Scientific and technical documents are increasingly written with software that allows a mixture of text and executable code, such as the Jupyter Notebook [KRKP+16], knitr [YX15], and Org-mode Babel [SD11], [SDDD12]. Writing with such tools can enhance reproducibility, simplify code documentation, and aid in automating reports.

This paper introduces Codebraid, which allows executable code within Pandoc Markdown documents [JG19], [JM19]. Codebraid is developed at https://github.com/gpoore/codebraid and is available from the Python Package Index (PyPI). It allows Markdown code blocks like the one below to be executed during the document build process. In this case, the "`.cb.run`" tells Codebraid to run the code and include the output.

````
```{.python .cb.run}
print("Running code within *Markdown!*")
```
````

The final document contains the code's output, interpreted as if it had been entered directly in the original Markdown source:

Running code within *Markdown!*

A document using Codebraid can be converted from Markdown into any of the many formats supported by Pandoc, such as HTML, Microsoft Word, LaTeX, and PDF. Codebraid delegates all Markdown parsing and format conversions to Pandoc, so it does not introduce any special restrictions on what is possible with a Pandoc Markdown document. This close integration with Pandoc also means that Codebraid can be extended in the future to work with additional document formats beyond Markdown.

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

Codebraid includes a built-in code execution system. It can also use Jupyter kernels [KRKP+16] to execute code. The first code block that is executed with a given language can specify a kernel. In the example below, the "`.cb.nb`" tells Codebraid to run the code and provide a "notebook" display that shows both code and output, while "`jupyter_kernel`" specifies a kernel.

````
```{.python .cb.nb jupyter_kernel=python3}
from sympy import *
init_printing()
x = Symbol('x')
integral = Integral(E**(-x**2), (x, -oo, oo))
display(integral)
integral.doit()
```
````

Because a Jupyter kernel was used to run the code, the result includes rich output in the form of rendered LaTeX math, just as it would in a Jupyter notebook:

```
from sympy import *
init_printing()
x = Symbol('x')
integral = Integral(E**(-x**2), (x, -oo, oo))
display(integral)
integral.doit()
```

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx$$

$$\sqrt{\pi}$$

The next section provides an example of the document build process with Codebraid. This is followed by an overview of Codebraid features and capabilities. Finally, the Comparison considers Codebraid in the context of knitr, Pweave, Org-mode Babel, and the Jupyter Notebook.

Building a simple Codebraid document

A simple Pandoc Markdown document that runs code with Codebraid is shown below.

````
```{.python .cb.run name=part1}
var1 = "Hello from *Python!*"
var2 = f"Here is some math:  $2^8={2**8}$."
```

```{.python .cb.run name=part2}
print(var1)
print(var2)
```
````

If this were a normal Pandoc document, converting it into a format such as reStructuredText could be accomplished by running

```
pandoc --from markdown --to rst file.md
```

Using Codebraid to execute code as part of the document conversion process is simply a matter of replacing `pandoc` with `codebraid pandoc`:

```
codebraid pandoc --from markdown --to rst file.md
```

The `codebraid` executable is available from the Python Package Index (PyPI); development is at https://github.com/gpoore/codebraid. By default, code is executed with Codebraid's built-in code execution system. This can easily be swapped for a Jupyter kernel, as shown in the Introduction and discussed in greater detail in Jupyter kernels.

When this `codebraid pandoc` command is executed, the original Markdown shown above is converted into Codebraid-processed Markdown:

```
Hello from *Python!*
Here is some math:  $2^8=256$.
```

This processed Markdown is then converted into the final reStructuredText, rendering as

Hello from *Python!* Here is some math: $2^8 = 256$.

By default, the output of code executed with `cb.run` is interpreted as Markdown. It is possible to show the output verbatim instead, as discussed later.

In this example, the code is simple enough that it could be executed every time the document is built, but that will often not be the case. By default, Codebraid caches all code output, and code is only re-executed when it is modified. This can be changed by building with the flag `--no-cache`.

Pandoc code attribute syntax

Pandoc Markdown defines an attribute syntax for inline code and code blocks. Codebraid uses this to designate which code blocks should be executed and provide options. Code attributes have the general form

```
{#id .class1 .class2 key1=value1 key2=value2}
```

If code with these attributes were converted into HTML, `#id` becomes an HTML id for the code, anything with the form `.class` specifies classes, and space-separated key-value pairs provide additional attributes. Although key-value pairs can be quoted with double quotation marks, Pandoc allows most characters except the space and equals sign unquoted. Other output formats such as LaTeX use attributes in a largely equivalent manner.

Pandoc uses the first class to determine the language name for syntax highlighting, hence the `.python` in the example in the last section. Codebraid uses the second class to specify a command for processing the code. All Codebraid commands are under a `cb` namespace to prevent unintentional collisions with normal Pandoc attributes. In the example, `cb.run` indicates that code should be run, stdout should be included and interpreted as Markdown, and stderr should be displayed in the event of errors. If a Jupyter kernel were in use, rich output such as plots would also be included. Finally, the `name` keyword is used to assign a unique name to each piece of code. This allows the code to be referenced elsewhere in a document to insert any combination of its Markdown source, code, stdout, stderr, and rich output (for Jupyter kernels).

Creating examples

The example in Building a simple Codebraid document was actually itself an example of using Codebraid. This paper was written in Markdown, then converted to reStructuredText via Codebraid with Pandoc. Finally, the reStructuredText was converted through LaTeX to PDF via Docutils [DG16]. The two code blocks in the example were only entered in the original Markdown source of this paper a single time, and Codebraid only executed them a single time. However, with Codebraid's copy-paste capabilities, it was possible to display the code and output at other locations in the document programmatically.

The rendered output of the two code blocks is shown at the very end of the earlier section. This is where the code blocks were actually entered in the original Markdown source of this paper, and where they were executed.

Recall that both blocks were given names, `part1` and `part2`. This enables any combination of their Markdown source, code, stdout, and stderr to be inserted elsewhere in the document. At the beginning of the earlier section, the Markdown source for the blocks was shown. This was accomplished via

````
```{.cb.paste copy=part1+part2 show=copied_markup}
```
````

The `cb.paste` command inserts copied data from one or more code chunks that are specified with the `copy` keyword. Meanwhile, the `show` keyword controls what is displayed. In this case, the Markdown source of the copied code chunks was shown. Since the `cb.paste` command is copying content from elsewhere, it is used with an empty code block. Alternatively, a single empty line or a single line containing an underscore is allowed as a placeholder.

Toward the end of the earlier section, the verbatim output of the Codebraid-processed Markdown was displayed. This was inserted in a similar manner:

````
```{.cb.paste copy=part1+part2 show=stdout:verbatim}
```
````

The default format of `stdout` is `verbatim`, but this was specified just to be explicit. The other option is `raw` (interpreted as Markdown).

Of course, all Markdown shown in the current section was itself inserted programmatically using `cb.paste` to copy from the earlier section. However, to prevent infinite recursion, the next section is not devoted to explaining how this was accomplished.

Other Codebraid commands

The commands `cb.run` and `cb.paste` have already been introduced. There are three additional commands.

The `cb.code` command simply displays code, like normal inline code or a code block. It primarily exists so that normal code can be named, and then accessed later. `cb.paste` could be used to insert the code elsewhere, perhaps combined with code from other sources via something like `copy=code1+code2`. It would also be possible to run the code elsewhere:

````
```{.cb.run copy=code1+code2}
```
````

When `copy` is used with `cb.run`, or another command that executes code, only code is copied, and everything proceeds as if this code had been entered directly in the code block.

The `cb.expr` command only works with inline code, unlike other commands. It evaluates an expression and then prints a string representation, which is interpreted as Markdown. For example,

```
`2**128`{.python .cb.expr}
```

produces

340282366920938463463374607431768211456

As this demonstrates, Pandoc code attributes for inline code immediately follow the closing backtick(s). While this sort of a "postfix" notation may not be ideal from some perspectives, it is the cost of maintaining full compatibility with Pandoc Markdown syntax.

Finally, the `cb.nb` command runs code and provides a "notebook" display. For inline code, `cb.nb` is like `cb.expr` except that it displays rich output or verbatim text. For code blocks, `cb.nb` displays code followed by verbatim stdout. If there are errors, stderr is also included automatically. When Codebraid is used with a Jupyter kernel, rich outputs such as plots are included as well. This was demonstrated in the Introduction.

Display options

There are two code chunk keywords that govern display, `show` and `hide`. These can be used to override the default display settings for each command.

`show` takes any combination of the following options: `markup` (display Markdown source), `code` (display code being executed), `stdout`, `stderr`, and `none`. There is also `rich_output` when a Jupyter kernel is used to execute code. Multiple options can be combined, such as `show=code+stdout+stderr`. Code chunks using `copy` can employ `copied_markup` to display the Markdown source of the copied code chunk. When the `cb.expr` command is used, the expression output is available via `expr`. Using `show` completely overwrites the existing display settings.

The display format can also be specified with `show`. For `stdout`, `stderr`, and `expr`, there are three formats: `raw` (interpreted as Markdown), `verbatim`, or `verbatim_or_empty` (verbatim if there is output, otherwise a space or empty line). For example, `show=stdout:raw+stderr:verbatim`. While a format can be specified for `markup` and `code`, only the default `verbatim` is permitted. For `rich_output`, the output representation (MIME type) can be selected. Thus, `show=rich_output:png` selects a PNG image representation.

`hide` takes the same options as `show`, except that `none` is replaced by `all` and formats are not specified. Instead of overriding existing settings like `show`, `hide` removes the specified display options from those that currently exist.

Codebraid code execution system

Codebraid currently provides two options for executing code: a built-in code execution system which is used by default and Jupyter kernels. Jupyter kernels are demonstrated in the next section. This section describes the built-in system, which currently supports Python 3.5+, Julia, Rust, R, Bash, and JavaScript. Any combination of these languages can be used within a single document. While the built-in system currently lacks Jupyter kernel features like rich output, it is nearly identical to extracting the code from a document, concatenating it, and executing it via the standard interpreter or compiler. As a result, it has low overhead and produces the same output as would have been generated by a separate source file.

Overview

The code from each code chunk is inserted into a template before execution. The template writes delimiters to stdout and stderr at the beginning of each code chunk. These delimiters are based on a hash of the code to avoid the potential for collisions. Once execution is complete, Codebraid parses stdout and stderr and uses these delimiters to associate output with individual code chunks. This system is a more advanced variant of the one I created previously in PythonTeX [GMP15].

By default, code must be divided into complete units. For example, a code block must contain an entire loop, or an entire function definition. (This restriction can be relaxed with the code-chunk keyword `complete`; see Incomplete units of code later.) If a code chunk is not complete (and this is not indicated), then the incomplete code will interfere with writing the delimiters.

To address this, each individual delimiter is unique, and is tracked individually by Codebraid. If incomplete code interferes with the template to produce an error, Codebraid can use any existing stderr delimiters plus parsing of stderr to find the source and generate an appropriate error message. If the code does not produce an error, but prevents a delimiter from being written or causes a delimiter to be written multiple times or not at the beginning of a line, this will also be detected and traced back. Under normal conditions, interfering with the delimiters without detection requires conscious effort.

Adding languages

Adding support for additional languages is simply a matter of creating the necessary templates and putting them in a configuration file. Basic language support can require very little, essentially just code for writing the delimiters to stdout and stderr. For example, Bash support is based on this three-line template:

```
printf "\n{stdout_delim}\n"
printf "\n{stderr_delim}\n" >&2
{code}
```

The Bash configuration file also specifies that the file extension `.sh` should be used, and provides another four lines of template code to enable `cb.expr`. So far, the longest configuration file, for Rust, is less than fifty lines—counting empty lines.

Stderr

Because code is typically inserted into a template for execution, if there are errors the line numbers will not correspond to those of the code that was extracted from the document, but rather to those of the code that was actually executed. Codebraid tracks line numbers during template assembly, so that executed line numbers can be converted into original line numbers. Then it parses stderr and corrects line numbers. An example of an error produced with `cb.nb` with Python is shown below. Notice that the line number displayed is correct.

```
var = 123
print(var, flush=True)
var += "a"


123


Traceback (most recent call last):
  File "source.py", line 3, in <module>
    var += "a"
TypeError: unsupported operand type(s) for +=:
'int' and 'str'
```

Since line numbers in errors and warnings correspond to those in the code entered by the user, and since anything written to stderr is displayed by default next to the code that caused it, debugging is significantly simplified. In many cases, this even applies to compile errors for a language like Rust.

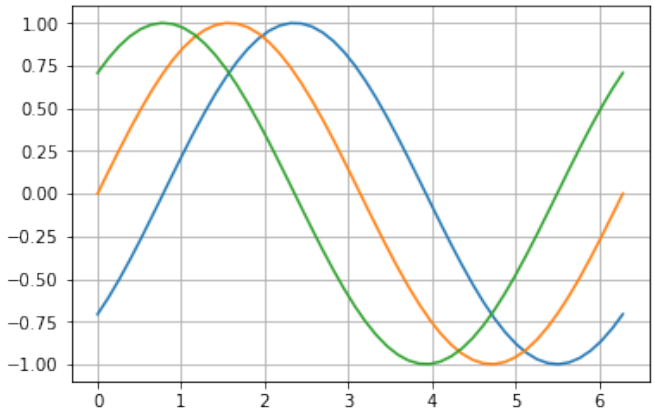
Jupyter kernels

Using a Jupyter kernel instead of the built-in code execution system is as simple as adding `jupyter_kernel=<name>` to the first code chunk for a language (or, as discussed later, to the first code chunk of a named `session`):

````
```{.python .cb.run jupyter_kernel=python3}
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
```

```{.python .cb.run}
x = np.linspace(0, 2*np.pi)
for n in range(-1, 2):
    plt.plot(x, np.sin(x + n*np.pi/4))
plt.grid()
```
````

Notice that `jupyter_kernel` was only needed (and only allowed) for the first code chunk. The second code chunk is still using the same language (`.python`), so it shares the same kernel. This Markdown results in a plot, just as it would within a Jupyter notebook. Because `cb.run` was used rather than `cb.nb`, code is not displayed and only the plot is shown:



The built-in code execution system allows multiple languages within a single document. This is also possible when Jupyter kernels are used instead. A single document can involve multiple kernels. Multiple independent sessions for the same kernel type are also possible when `jupyter_kernel` is combined with `session` (described in the next section). Of course, kernel features like IPython magics [IDT19a] are fully functional as well.

Advanced code execution

Ideally, executable code should be arranged within a document based on what is best for the reader, rather than in a manner dictated by limitations of the tooling. Several options are provided to maximize the flexibility of code presentation.

Incomplete units of code

By default, Codebraid requires that code be divided into complete units. For example, a code block must contain an entire loop, or an entire function definition. Codebraid's built-in code execution system can detect the presence of an incomplete unit of code because it interferes with stdout and stderr processing, in which case Codebraid will raise an error. Attempting to run an incomplete unit of code with a Jupyter kernel will also result in an error.

The `complete` keyword allows incomplete units of code. While this increases the flexibility of code layout, it also means that any output will not be shown until the next complete code chunk.

The Markdown for a somewhat contrived example that demonstrates these capabilities is shown below, along with its output. While this example uses Codebraid's code execution system, exactly the same result is obtained by using a Jupyter kernel.

````
```{.python .cb.run complete=false}
for n in range(11):
    if n % 2 == 0:
```

```{.python .cb.run}
        if n < 10:
            print(f"{n}, ", end="")
        else:
            print(f"{n}")
```
````

0, 2, 4, 6, 8, 10

Sessions

By default, all code for a language is executed within a single default session, so variables and data are shared between code chunks. It can be convenient to separate code into multiple sessions when several independent tasks are being performed, or when a long calculation is required but the output can easily be saved and loaded by separate code for visualization or other processing. The `session` keyword makes this possible. Session names are restricted to valid Python identifiers. For example,

````
```{.python .cb.run session=long}
import json
result = sum(range(100_000_000))
with open("result.json", "w") as f:
    json.dump({"result": result}, f)
```
````

Sessions work with both Codebraid's built-in code execution system and Jupyter kernels. For example, it is possible to have multiple independent sessions with a `python3` kernel within a single document.

All sessions are currently executed in serial. In the future, support for parallel execution may be added.

Outside `main()`

Codebraid's built-in code execution system runs code by inserting it into a template. The template allows stdout and stderr to be broken into pieces and correctly associated with the code chunks that created them. For a language like Python under typical usage, `complete` eliminates the few limitations of this approach. However, the situation for a compiled language with a `main()` function is more complex.

Codebraid includes support for Rust. By default, code is inserted into a template that defines a `main()` function. Thus, a code block like

````
```{.rust .cb.run}
let x = "Greetings from *Rust!*";
```
````

```
println!("{}", x);
```

can run to produce

> Greetings from *Rust!*

In some situations, it would be convenient to completely control the definition of the `main()` function and add code outside of `main()`. The `outside_main` keyword makes this possible. All code chunks with `outside_main=true` at the beginning of a session are used to overwrite the beginning of the `main()` template (everything before `main()`), while any chunks with `outside_main=true` at the end of the session are used to overwrite the end of the `main()` template (everything after `main()`). If all code chunks have `outside_main=true`, then all of Codebraid's templates are completely omitted, and all output is associated with the final code chunk. The example below demonstrates this option.

```
```{.rust .cb.run outside_main=true}
fn main() {
    use std::fmt::Write as FmtWrite;
    use std::io::Write as IoWrite;
    let x = "Rust says hello.  Again!";
    println!("{}", x);
}
```
```

> Rust says hello. Again!

Working with external files

Though Codebraid is focused on embedding executable code within a document, there will be times when it is useful to interact with external source files. Since Codebraid's built-in code execution system processes code with a programming language's standard interpreter or compiler, normal module systems are fully compatible; for example, in Python, `import` works normally. Of course, this is also true when working with Jupyter kernels. Codebraid provides additional ways to work with external files via the `include_file` option.

When `include_file` is used with the `cb.code` command, an external source file is simply included and displayed. It is possible to include only certain line ranges using the additional option `include_lines`, or only part of a file that matches a regular expression via `include_regex`. For example,

```
```{.markdown .cb.code include_file=poore.txt
include_regex="# Working.*?,"}
```
```

includes the original Markdown source for this paper, and then uses a regular expression to display only the first few lines of this current section on working with external files:

```
# Working with external files

Though Codebraid is focused on embedding executable
code within a document,
```

Since the `cb.code` command is including content from elsewhere, it is used with an empty code block. Alternatively, a single empty line or a single line containing an underscore is allowed as a placeholder.

This example included part of a file using a single regular expression. There are also options for including everything starting with or starting after a literal string or regular expression, and for including everything before or through a literal string or regular expression.

The `include_file` option works with commands that execute code as well. For instance,

```
```{.python .cb.run include_file=code.py}
```
```

would read in the contents of an external file "code.py" and then run it in the default Python session, just as if it had been entered directly within the Markdown file.

Comparison

To put Codebraid in context, this section provides a comparison with knitr, Pweave, Org-mode Babel, and the Jupyter Notebook. The comparison focuses on the default features of each program. Extensions for these programs and additional programs with similar features are summarized in the Appendix.

knitr

knitr [YX15] provides powerful R evaluation in Markdown, LaTeX, HTML, and other formats. It was inspired by Sweave [FL02], which allows R in LaTeX. The reticulate [AUT+19] and JuliaCall [CL19] packages for R have given knitr significant Python and Julia capabilities as well, including the ability to convert objects between languages. knitr is commonly used with the RStudio IDE, which provides a two-panel source-and-output preview interface as well as a notebook-style mode with inline display of results.

While knitr provides superior support for R, Codebraid focuses on providing more capabilities for other languages. knitr runs all R, Python, and Julia code in language-specific sessions, so data and variables are shared between code chunks. For all other languages, each code chunk is run in a separate process and there is no such continuity. Codebraid's built-in code execution system is designed to allow any language to share a session between multiple code chunks, and Jupyter kernels provide equivalent capabilities. R, Python, and Julia are limited to a single shared session each with knitr. Codebraid allows multiple sessions for all supported languages. This allows independent computations to be divided into separate sessions and only re-executed when necessary.

Once code is executed, Codebraid and knitr provide similar basic features for displaying the code and its output. knitr has more advanced options for formatting output, such as customizing plot appearance, converting plots into figures with captions, or combining plots into an animation.

The two programs take different approaches to extracting code from Markdown documents. knitr uses the custom R Markdown [RSt18] syntax to designate code that should be executed. It extracts inline code and code blocks from the original Markdown source using a preprocessor, then inserts the code's output into a copy of the document that can subsequently be processed with Pandoc. Because the preprocessor is based on simple regex matching, it does not understand Markdown comments and will run code in a commented-out part of a document. Writing tutorials that show literal knitr code chunks can involve inserting empty strings, zero-width spaces, line breaks, or Unicode escapes to avoid the preprocessor's tendency to execute code [YX19], [Hov17]. With Codebraid, Pandoc is used to convert a Markdown document into Pandoc's abstract syntax tree (AST) representation. Code extraction and output insertion are performed as operations on the

AST, and then Pandoc converts the modified AST into the final output document. This has the advantage that Pandoc handles all parsing and conversion, at the cost of running Pandoc multiple times.

Pweave

Pweave [MP16] is inspired by Sweave [FL02] and knitr [YX15], with a focus on Python in Markdown and other formats like LaTeX and reStructuredText. Pweave uses a custom Markdown syntax similar to knitr's for designating code blocks that should be executed, with many similar features and options. It also extracts code from Markdown documents with a simple preprocessor. Code is executed with a single Jupyter kernel. Any kernel can be used; the default is `python3`. Rich output like plots can be included automatically.

Like knitr, Pweave provides some more advanced options for display formatting that Codebraid lacks, primarily related to figures. Codebraid has advantages in three areas. Code execution is more flexible since it allows multiple Jupyter kernels per document and multiple independent sessions per kernel, in addition to the built-in code execution system. Since Codebraid uses Pandoc for all Markdown parsing, it avoids the limitations of a preprocessor. Codebraid also provides a broader set of display capabilities, including the ability to programmatically copy and display code or its output into other parts of a document.

Org-mode Babel

Babel [SD11], [SDDD12] allows code blocks and inline code in Emacs Org-mode documents to be executed. Any number of languages can be used within a single document. By default, each code chunk is executed individually in its own process. For many interpreted languages, it is also possible to run code in a session so that data and variables persist between code chunks. In those cases, multiple sessions per language are possible. Any combination of code and its stdout can be displayed. Stdout can be shown verbatim or interpreted as Org-mode, HTML, or LaTeX markup. For some languages, such as gnuplot, graphical output can also be captured and included automatically.

Babel can function as a meta-programming language for Org mode. A code chunk can be named, and then a later code chunk—potentially in a different language—can access its output by name and perform further processing. Similarly, there are literate programming capabilities that allow a code chunk to copy the source of one or more named chunks into itself, essentially serving as a template, before execution.

Codebraid is like a Markdown-based Babel with additional code execution capabilities but without some of the meta-programming and literate programming options. Codebraid allows sessions for all languages, not just for some interpreted languages. It provides broad support for rich output like plots through Jupyter kernels. Stderr can also be displayed. While Codebraid currently lacks a system for passing output between code chunks, it does provide some literate-programming style capabilities for code reuse.

Jupyter Notebook

The Jupyter (formerly IPython) Notebook [KRKP+16] provides a browser-based user interface in which a document is represented as a series of cells. A cell may contain Markdown (which is converted into HTML and displayed when not being edited), raw text, or code. Code is executed by language-specific backends, or kernels. Well over one hundred kernels are available beyond Python, including Julia, R, Bash, and even compiled languages like C++ and Rust [Jup19c]. Jupyter kernels are often used with the Jupyter Notebook, but they can also function as a standalone code execution system.

A Jupyter Notebook can only have a single kernel, and thus only a single primary programming language with a single session or process. This means that dividing independent computations into separate sessions or processes is typically not as straightforward as it might be in Org-mode Babel or Codebraid. However, the interactive nature of the notebook often reduces the impact of this limitation, and can actually be a significant advantage. Code cells can be run one at a time; a single code cell can be modified and run again without re-executing any previous code cells.

Some kernels include support for interacting with additional languages. The IPython kernel [IDT19b] has `%%script` and similar "magics" [IDT19a] that allow single cells to be executed in a subprocess by another language. PyJulia [JIdt19] and rpy2 [LGrc16] provide more advanced magics that allow an IPython kernel to interact with a single Julia or R session over a series of cells (see [MB18b] for examples).

While Codebraid lacks the Jupyter Notebook's interactivity, it does have several capabilities not present in the default Notebook. A Codebraid document can involve multiple Jupyter kernels, as well as multiple independent sessions per kernel. It can execute both code blocks and inline code; the Jupyter Notebook is limited to executing code in code cells. Code layout is more flexible with Codebraid because a code chunk can contain an incomplete unit of code, such as part of a loop or part of a function definition. This is possible even when working with Jupyter kernels. Codebraid also provides more flexible display options. It is possible to show any combination of code, stdout, stderr, or rich output in any order, and to select which form of rich output (MIME type) is shown. Code or its output can be copied programmatically, so code can be executed at one location in a document and its output displayed elsewhere.

Conclusion

Codebraid provides a unique and powerful combination of features for executing code embedded in Pandoc Markdown documents.

- Both code blocks and inline code can be executed.
- Code blocks are not required to contain complete units of code, like a complete loop or function definition.
- A single document can use multiple languages and multiple independent sessions per language. Any language can share a session between multiple code chunks. Independent computations can be divided into separate sessions and only re-executed when necessary.
- Code can be executed with the built-in system, or with Jupyter kernels which provide rich output such as plots.
- A code chunk can display any combination of its Markdown source, code, stdout, stderr, and rich output.
- It is easy to reuse code and its output programmatically with the paste functionality. It is also possible to include all or part of an external source file for display or execution.
- Because only standard Pandoc Markdown syntax is used, all Markdown parsing and document conversion can be delegated to Pandoc, and there are no issues with preprocessors that do not fully support Markdown syntax.

There are several logical avenues for further development. One of the original motivations for creating Codebraid was to build on my previous work with PythonTeX [GMP15] to create a program that could be used with multiple markup languages. While Codebraid has focused thus far on Pandoc Markdown, little of it is actually Markdown-specific. It should be possible to work with other markup languages supported by Pandoc, such as LaTeX; all that is required is that Pandoc parses key-value attributes for some variant of a code block. Pandoc has recently added Jupyter notebooks to its extensive list of supported formats. Perhaps at some point it will be possible to convert a Codebraid document into a Jupyter notebook, perform some exploratory programming for a single session of a single language, and then convert back to Markdown.

Codebraid's caching system could also be improved in the future. Currently, caching is based only on the code that is executed. Adding a way to specify external dependencies such as data files would be beneficial.

Appendix

The Comparison focuses on the default features of knitr, Pweave, Org-mode Babel, and the Jupyter Notebook. This appendix summarizes extensions for these programs and additional programs with similar features.

knitr extensions

Though knitr does not include any support for Jupyter kernels, the knitron [FH16] and ipython_from_R [MW18b] packages have demonstrated that this is technically feasible.

Software similar to Pweave

The Comparison includes Pweave [MP16] because it is one of the most capable knitr-like systems for other languages. There are several other similar programs.

Weave.jl [MP17], by the creator of Pweave, provides similar features for executing Julia code. It uses Julia to manage code execution rather than a Jupyter kernel.

knitpy [Kat18] describes itself as a port of knitr to Python. It uses knitr-style Markdown syntax, and provides code-block options to control basic code and output display. Other knitr-style options are not supported. Code is executed in a single Jupyter IPython kernel. stitch [TA16] is similar, drawing inspiration from knitr and knitpy. Compared to knitpy, it lacks options for customizing output display but has options for customizing figure display.

Knitj [JH19] is another Jupyter kernel–Markdown integration. Options for controlling display are contained in special comments in the first line of code within a code block, rather than in the code block's Markdown attributes. It focuses on producing HTML and includes efficient live preview capabilities.

There are also some comparable tools for reStructuredText. nb2plots can convert an ipynb notebook file into reStructuredText for Sphinx [MB18a]. When Sphinx builds the document, the code is still executed and plots are automatically included, so the live code and rich output of the notebook are not lost. It is possible to customize display by hiding code. The reStructuredText can also be converted to a Python source file or ipynb when that is desired.

The Jupyter Sphinx Extension [Jup19b] provides a `jupyter-execute` directive for running code in a Jupyter kernel. By default, code is executed within a single kernel, providing continuity. It is also possible to switch to a different kernel or switch to a different session using the same kernel type. Code and output (including rich output like plots) are displayed by default, but there are options for hiding code or output, or reversing their order. All code for a given Jupyter session can be converted into a script or a Jupyter notebook.

Org-mode Babel extensions

Packages like ob-ipython [GS17] and emacs-jupyter [NN19] allow Jupyter kernels [KRKP+16] instead of Babel's built-in code execution system. These add the capability to display error messages or rich output like graphics. The Emacs IPython Notebook [JMM19] takes a different approach by providing a complete Jupyter Notebook client in Emacs.

Jupyter Notebook extensions and related software

Some more general approaches to working around the limitation of one kernel per notebook are provided by the BeakerX polyglot magics [TSOS18], which support bidirectional autotranslation of data between languages, and the Script of Scripts (SoS) kernel [BP19], which acts as a managing kernel over multiple normal kernels.

It is possible to execute inline code within Markdown cells with the Python Markdown extension [Jup18c]. This treats Markdown cells as `{{expression}}`-style templates so long as inline code is outside LaTeX equations. The extension also supports notebook export to other document formats with nbconvert [Jup19a] via a bundled preprocessor.

The Comparison does not consider hiding code or output in documents derived from Jupyter notebooks because this is possible with nbconvert [Jup19a] as well as extensions and other programs. Hiding code or output in exported documents is possible on a notebook-wide basis by configuring nbconvert with the `TemplateExporter` exclude options. It is also possible at the individual cell level by adding a tag to a cell (View, Cell Toolbar, Tags, then "Add tag") and then configuring nbconvert to use the desired `TagRemovePreprocessor` with a given tag. An alternative is to use extensions with their provided preprocessors or templates [Jup18a], [Jup18b], or employ a more comprehensive tool like Jupyter Book [LH19] that defines a set of tags for display customization.

The Comparison does not cover the Jupyter Notebook's JSON-based ipynb file format because there are multiple ways to work around its limitations. There are special diffing tools for ipynb files such as nbdime [MSA15]. It is also possible to save notebooks as Markdown files instead, or convert them to source code with Markdown in comments:

- Jupytext [MW18a], [MWtJT19] can convert Jupyter notebooks into Markdown or R Markdown (knitr), or into scripts in which code cells are converted into code while Markdown cells are converted into intervening comments. These formats can also be converted into Jupyter notebooks.
- notedown [AO16] can convert between Markdown and ipynb, and can also work with R Markdown documents.
- Pandoc [JM19] can convert to or from ipynb files. Notebooks, including cells along with their attributes, can be

represented as standard Pandoc Markdown. podoc [CR18] is an earlier program for converting between ipynb and Pandoc's AST. It builds on the prior ipymd [CR16].

- The Hydrogen package [Hyd19] for the Atom text editor provides conversion between ipynb and source code plus comments. When such a source file is edited, Hydrogen can connect to a Jupyter kernel to display rich output inline within the editor. Similar capabilities are provided by the Python extension for VS Code [Mic19].

Of the programs listed above, Jupytext, notedown, and podoc provide `ContentsManager` subclasses for the Jupyter Notebook that allow it to seamlessly use Markdown as a storage format.

References

[AO16] Aaron O'Leary. Convert IPython Notebooks to markdown (and back), 2016. URL: https://github.com/aaren/notedown.

[AUT+19] JJ Allaire, Kevin Ushey, Yuan Tang, Dirk Eddelbuettel, Bryan Lewis, and Marcus Geelnard. reticulate: R Interface to Python, 2019. URL: https://rstudio.github.io/reticulate/index.html.

[BP19] Bo Peng. SoS: Notebook environment for both interactive data analysis and batch data processing, 2019. URL: https://vatlab.github.io/sos-docs/.

[CL19] Changcheng Li. JuliaCall: an R package for seamless integration between R and Julia. *The Journal of Open Source Software*, 4(35):1284, 2019. doi:10.21105/joss.01284.

[CR16] Cyrille Rossant. Replace .ipynb with .md in the IPython Notebook, 2016. URL: https://github.com/rossant/ipymd.

[CR18] Cyrille Rossant. podoc, 2018. URL: https://github.com/podoc/podoc.

[DG16] David Goodger. Docutils Project Documentation Overview, 2016. URL: http://docutils.sourceforge.net/docs/index.html.

[FH16] Fabian Hirschmann. knitron: knitr + IPython + matplotlib, 2016. URL: https://github.com/fhirschmann/knitron/.

[FL02] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9. URL: http://www.stat.uni-muenchen.de/~leisch/Sweave.

[GMP15] Geoffrey M. Poore. PythonTeX: reproducible documents with LaTeX, Python, and more. *Computational Science & Discovery*, 8(1):014010, July 2015. URL: https://doi.org/10.1088%2F1749-4699%2F8%2F1%2F014010, doi:10.1088/1749-4699/8/1/014010.

[GS17] Greg Sexton. Readme, 2017. URL: https://github.com/gregsexton/ob-ipython.

[Hov17] T. Hovorka. How to Show R Inline Code Blocks in R Markdown, 2017. URL: https://rviews.rstudio.com/2017/12/04/how-to-show-r-inline-code-blocks-in-r-markdown/.

[Hyd19] Hydrogen Contributors. Hydrogen, 2019. URL: https://nteract.gitbooks.io/hydrogen/.

[IDT19a] The IPython Development Team. Built-in magic commands, 2019. URL: https://ipython.readthedocs.io/en/stable/interactive/magics.html.

[IDT19b] The IPython Development Team. IPython Documentation, 2019. URL: https://ipython.readthedocs.io/en/stable/index.html.

[JG19] John Gruber. Markdown, 2002–2019. URL: https://daringfireball.net/projects/markdown/.

[JH19] Jan Hermann. Knitj, 2019. URL: https://github.com/jhrmnn/knitj.

[JIdt19] The Julia and IPython development teams. Welcome to PyJulia's documentation!, 2019. URL: https://pyjulia.readthedocs.io.

[JM19] John MacFarlane. Pandoc: a universal document converter, 2006–2019. URL: https://pandoc.org/.

[JMM19] John M. Miller. The Emacs IPython Notebook, 2019. URL: http://millejoh.github.io/emacs-ipython-notebook/.

[Jup18a] Jupyter Contrib Team. Codefolding, 2015–2018. URL: https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/codefolding/readme.html.

[Jup18b] Jupyter Contrib Team. Hide Input, 2015–2018. URL: https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/hide_input/readme.html.

[Jup18c] Jupyter Contrib Team. Unofficial Jupyter Notebook Extensions: Python Markdown, 2015–2018. URL: https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/python-markdown/readme.html.

[Jup19a] Jupyter Development Team. nbconvert: Convert Notebooks to other formats, 2015–2019. URL: https://nbconvert.readthedocs.io.

[Jup19b] Jupyter Development Team. Jupyter Sphinx Extension, 2019. URL: https://jupyter-sphinx.readthedocs.io.

[Jup19c] Jupyter Team. Jupyter kernels, 2019. URL: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

[Kat18] Jan Katins. knitpy: Elegant, flexible and fast dynamic report generation with python, 2018. URL: https://github.com/jankatins/knitpy.

[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, and Carol Willing. Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016. doi:10.3233/978-1-61499-649-1-87.

[LGrc16] Laurent Gautier & rpy2 contributors. Documentation for rpy2, 2008–2016. URL: https://rpy2.readthedocs.io.

[LH19] Sam Lau and Chris Holdgraf. Jupyter Book: Books with Jupyter and Jekyll, 2019. URL: https://jupyter.org/jupyter-book/intro.

[MB18a] Matthew Brett. nb2plots - the documentation that is not missing, 2016–2018. URL: http://matthew-brett.github.io/nb2plots/.

[MB18b] Matthias Bussonnier. I Python, You R, We Julia, 2018. URL: https://blog.jupyter.org/i-python-you-r-we-julia-baf064ca1fb6.

[Mic19] Microsoft. Working with Jupyter Notebooks in Visual Studio Code, 2019. URL: https://code.visualstudio.com/docs/python/jupyter-support.

[MP16] Matti Pastell. Pweave - Scientific Reports Using Python, 2010–2016. URL: http://mpastell.com/pweave/.

[MP17] Matti Pastell. Weave.jl: Scientific Reports Using Julia. *Journal of Open Source Software*, 2(11), 2017. URL: http://joss.theoj.org/papers/10.21105/joss.00204, doi:10.21105/joss.00204.

[MSA15] Martin Sandve Alnæs. nbdime – diffing and merging of Jupyter Notebooks, 2015. URL: https://nbdime.readthedocs.io/en/latest/index.html.

[MW18a] Marc Wouts. Introducing Jupytext, 2018. URL: https://towardsdatascience.com/introducing-jupytext-9234fdff6c57.

[MW18b] Marc Wouts. ipython_from_R: Communicate with jupyter kernels from R, 2018. URL: https://github.com/mwouts/ipython_from_R.

[MWtJT19] Marc Wouts and the Jupytext Team. Jupyter notebooks as Markdown documents, Julia, Python or R scripts, 2018–2019. URL: https://jupytext.readthedocs.io/.

[NN19] Nathaniel Nicandro. An interface to communicate with Jupyter kernels in Emacs, 2019. URL: https://github.com/dzop/emacs-jupyter.

[RSt18] RStudio Inc. R Markdown, 2016–2018. URL: https://rmarkdown.rstudio.com/.

[SD11] E. Schulte and D. Davison. Active Documents with Org-Mode. *Computing in Science Engineering*, 13(3):66–73, May 2011. doi:10.1109/MCSE.2011.41.

[SDDD12] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A Multi-Language Computing Environment for Literate Programming and Reproducible Research. *Journal of Statistical Software*, 46(3):1–24, 1 2012. URL: http://www.jstatsoft.org/v46/i03.

[TA16] Tom Augspurger. stitch, 2016. URL: https://pystitch.github.io/.

[TSOS18] LLC Two Sigma Open Source. BeakerX, 2018. URL: http://beakerx.com/.

[YX15] Yihui Xie. *Dynamic Documents with R and knitr*. Chapman & Hall/CRC Press, 2015.

[YX19] Yihui Xie. Frequently asked questions, 2005–2019. URL: https://yihui.name/knitr/faq/.

Solving Polynomial Systems with phcpy

Jasmine Otto[‡*], Angus Forbes[‡], Jan Verschelde[§]

Abstract—The solutions of a system of polynomials in several variables are often needed, e.g.: in the design of mechanical systems, and in phase-space analyses of nonlinear biological dynamics. Reliable, accurate, and comprehensive numerical solutions are available through PHCpack, a FOSS package for solving polynomial systems with homotopy continuation.

This paper explores new developments in phcpy, a scripting interface for PHCpack, over the past five years. For instance, phcpy is now available online through a JupyterHub server featuring Python2, Python3, and SageMath kernels. As small systems are solved in real-time by phcpy, they are suitable for interactive exploration through the notebook interface. Meanwhile, phcpy supports GPU parallelization, improving the speed and quality of solutions to much larger polynomial systems. From various model design and analysis problems in STEM, certain classes of polynomial system frequently arise, to which phcpy is well-suited.

Introduction

The Python package phcpy [Ver14] provides an alternative to the command line executable `phc` of PHCpack [Ver99] to solve polynomial systems by homotopy continuation methods. In the phcpy interface, Python scripts replace command line options and text menus, and data persists in a session without temporary files. This also makes PHCpack accessible from Jupyter notebooks, including a JupyterHub server available online [Pascal].

phcpy takes as input a list of polynomials in several variables, with complex-valued floating-point coefficients. Homotopy methods connect this given system to a 'start system' with known solutions. A homotopy is a family of polynomial systems where one of the variables is considered as a parameter. Polynomial homotopy continuation combines the application of homotopy and continuation methods, which extend the convergence of Newton's method from local to global, to solve polynomial systems.

Numerical continuation methods track the solution paths, depending on the parameter, originating at the known solutions to the solutions of the given system. phcpy is also able to represent the numerical irreducible decomposition of the system's solution set, which yields the *positive dimensional solution sets* containing infinitely many points, in addition to the isolated solutions.

The focus of this paper is on the application of new technology to solve polynomial systems, in particular, cloud computing [BSVY15] and multicore shared memory parallelism accelerated with graphics processing units [VY15]. Our web interface offers phcpy in a SageMath [Sage], [SJ05] kernel or in a Python kernel of a Jupyter notebook [Klu16].

Although phcpy has been released for only five years, three instances in the research literature of symbolic computation, geometry and topology, and chemical engineering (respectively) mention its application to their computations.

- The number of embeddings of minimally rigid graphs [BELT18].
- Roots of Alexander polynomials [CD18].
- Critical points of equilibrium problems [SWM16].

The package phcpy is in ongoing development. At the time of writing, this paper is based on version 0.9.5 of phcpy, whereas version 0.1.5 was current at the time of [Ver14]. An example of these changes is that the software described in [SVW03] was recently parallelized for phcpy [Ver18].

A Scripting Interface for PHCpack

The mission of phcpy is to bring polynomial homotopy continuation into Python's computational ecosystem.

The package phcpy wraps the shared object files of a compiled PHCpack, which makes the methods more accessible without sacrificing their efficiency. First, the wrapping transfers the implementation of the many available homotopy algorithms in a direct way into Python modules. Second, we do not sacrifice the efficiency of the compiled code. Scripts replace the input/output movements and interactions with the user, but not the computationally intensive algorithms.

Numerical algebraic geometry [SVW05] was introduced in 1995 as a pun on numerical linear algebra. PHCpack prototyped the first algorithms to compute a numerical irreducible decomposition of the solution set of a polynomial system. The package phcpy aims to bring the algorithms of numerical algebraic geometry into the computational ecosystem of Python.

Related Software

PHCpack is one of three FOSS packages for polynomial homotopy computation currently under development. Of these, only Bertini 2 [Bertini2.0] also offers Python bindings, although it is not GPU-accelerated and does not export the numerical irreducible decomposition, among other differences. Version 1.4 of Bertini is described in [BHSW13].

HomotopyContinuation.jl [HCJL] is a standalone package for Julia, presented at ICMS 2018 [BT18].

NAG4M2 [NAG4M2] is a package for Macaulay2 (a standard computational algebra system [M2]), which can also act an interface to PHCpack or Bertini. As described in [Ley11], it provided the starting point for PHCpack's Macaulay2 bindings [GPV13].

* *Corresponding author: jtotto@ucsc.edu*
‡ *University of California, Santa Cruz*
§ *University of Illinois at Chicago*

User Interaction

Online Access

The first area of improvement that phcpy brings is in the interaction with the user.

With JupyterHub [JUPH], we provide online access [Pascal] to environments with Python and SageMath kernels pre-installed, both featuring phcpy and tutorials on its use (per next section). Since Jupyter is language-agnostic, execution environments in several dozen languages are possible. Our users can also run code in a Python Terminal session. As of the middle of May 2019, our web server has 146 user accounts, each having access to our JupyterHub instance. Our server is available for public use, after creating a free account.

In our first design of a web interface to `phc`, we developed a collection of Python scripts (mediated through HTML forms), following common programming patterns [Chu06]. This design is described in Chapter 6 of [Yu15]. For the user administration of our JupyterHub, we refreshed this first web interface, retaining the following architecture.

MySQLdb [MSDB] does the management of user data, including a) names and encrypted passwords, b) generic, random folder names to store data files, and c) file names with polynomial systems they have solved. With the module smtplib, we defined email exchanges for an automatic 2-step registration process and password recovery protocol.

A custom JupyterHub Authenticator connects to the existing MySQL database and triggers a SystemdSpawner that isolates the actions of users to separate processes and logins in generic home folders. The email account management prompts were hooked to new Tornado RequestHandler instances, which perform account registration and activation in the database, as well as password recovery and reset. Each such route serves HTML forms seamlessly with the JupyterHub interface, by extending its Jinja templates.

Code Snippets

Learning a new API is daunting enough without also being a crash course in algebraic geometry. Therefore, the user's manual of phcpy [PHCPY] begins with a tutorial section using only the blackbox solver `phcpy.solver.solve(system, ...)`. In this API, `system` is a list of strings representing polynomials, terminated by semicolons, and containing as many variables as equations.

The code snippets from these tutorials are available in our JupyterHub deployment, via the snippets menu provided by nbextensions [JUP15]. This menu suggests typical applications to guide the novice user. The screen shot in Fig. 1 shows the code snippet reproduced below.

```
# PHCpy > blackbox solver > solving trinomials
#       > solving a specific case
from phcpy.solver import solve

f = ['x^2*y^2 + 2*x - 1;', 'x^2*y^2 - 3*y + 1;']
sols = solve(f)
for sol in sols: print(sol)
```

The first solution of the given trinomial can be read as (0.48613... + 0.0i, 0.34258... - 0.0i), where the imaginary part of x_0 is exactly zero, and that of y_0 negligibly small. Programmatically, these can be accessed using either `solve(f, dictionary_output=True)`, or equivalently by parsing strings through `[phcpy.solutions.strsol2dict(sol) for sol in solve(f)]`.

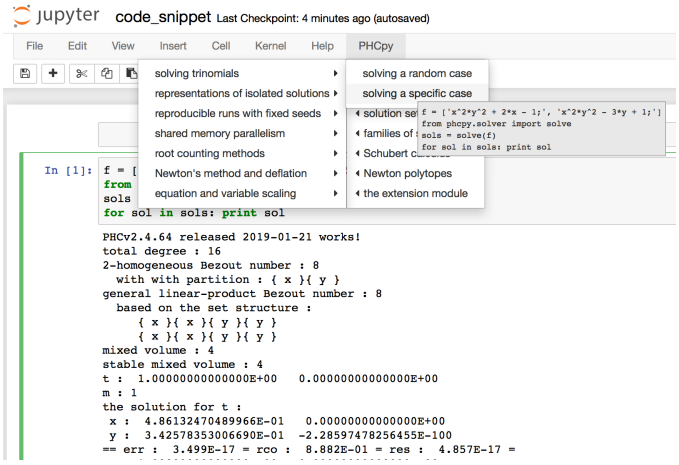


Fig. 1: *The code snippet for the blackbox solver.*

Direct Manipulation

One consequence of the Jupyter notebook's rich output is the possibility of rich input, as explored through ipywidgets [IPYW] and interactive plotting libraries. The combination of rich input with fast numerical methods makes surprising interactions possible, such as interactive solution of Apollonius' Problem, which is to construct all circles tangent to three given circles in a plane.

The tutorial given in the phcpy documentation was adapted for a demo accompanying a SciPy poster in 2017, whose code [APP] will run on our JupyterHub (by copying `apollonius_d3.ipynb` and `apollonius_d3.js` to one's own user directory).

This system of 3 nonlinear constraints in 5 parameters for each of 8 possible tangent circles can be solved interactively by our system in real-time (Fig. 2). Although any of the 8 tangent circles could have nonzero imaginary part in their x/y position or radius, depending on input coefficients (input circles), such circles are not rendered. Thanks to its rich output capabilities, Jupyter is a suitable environment for mapping algebraic inputs to the planar geometric objects they represent (a data binding) through D3.js [D3].

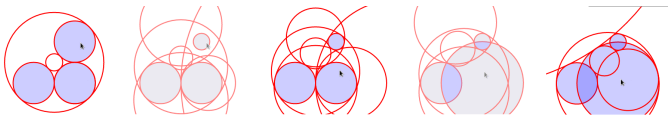


Fig. 2: *Tangent circles calculated by phcpy in response to user reparameterization of the system.*

This approach makes use of the real-time solution of small polynomial systems, demonstrating the low latency of phcpy. It complements static input conditions by investigating their continous deformation, especially across singular solutions (which PHCpack handles more robustly than naive homotopy methods). Singular solutions of polynomial systems are handled by deflation [LVZ06], which restores quadratic convergence of Newton's method by the addition of sufficiently many higher order derivatives to the original system.

Solving Polynomial Systems

Our input is a list of polynomials in several variables. This input list represents a polynomial system. By default, the coefficients of

the polynomials are considered as complex floating point numbers. The system is then solved over the field of complex numbers.

For general polynomial systems, the complexity of the solution set can be expected to grow exponentially in the dimensions (number of polynomials and variables) of the system. The complexity of computing all solutions of a polynomial system is #P-hard. The complexity class #P is the class of counting problems. Formulating instances of polynomial systems that will occupy fast computers for a long time is not hard.

Polynomial Homotopy Continuation

By computing over the field of complex numbers, we exploit the continuity of the solution set as a function of the coefficients of the polynomials in the system. These numerical algorithms, called continuation methods, track solution paths defined by a one parameter family of polynomial systems (the homotopy). Homotopy methods take a polynomial system as input, and construct a suitable embedding of the input system into a family which contains a start system with known solutions.

We say that a homotopy is *optimal* if for generic instances of the coefficients of the input system no solution paths diverge. Even as the complexity of the solution set is very hard, the problem of computing the next solution, or just one random solution, has a much lower complexity. phcpy offers optimal homotopies for three classes of polynomial systems:

1) *dense polynomial systems*
   A polynomial of degree d can be deformed into a product of d linear polynomials. If we do this for all polynomials in the system (as in [VC93]), then the solutions of the deformed system are solutions of linear systems. Continuation methods track the paths originating at the solutions of the deformed system to the given problem.
2) *sparse polynomial systems*
   A system is sparse if relatively few monomials appear with nonzero coefficient. The convex hulls of the exponent vectors of the monomials that appear are called Newton polytopes. The mixed volume of the tuple of Newton polytopes associated with the system is a sharp upper bound for the number of isolated solutions. Polyhedral homotopies ([HS95], [VVC94]) start at solutions of systems that are sparser than the given system and extend those solutions to the solutions of the given problem.
3) *Schubert problems in enumerative geometry*
   The classical example is to compute all lines in 3-space that meet four given lines nontrivially. Homotopies to solve geometric problems move the input data to special position, solve the special configuration, and then deform the solutions of the special problem into those of the original problem. Such homotopies were introduced in [HSS98].

All classes of homotopies share the introduction of random constants.

For its fast mixed volume computation, the software incorporates MixedVol [GLW05] and DEMiCs [MT08]. High-precision double double and quad double arithmetic is performed by the algorithms in QDlib [HLB01].

Speedup and Quality Up

The solution paths defined by polynomial homotopies can be tracked independently, providing obvious opportunities for parallel execution. This section reports on computations on our server, a 44-core computer.

An obvious benefit of running on many cores is the speedup. The *quality up* question asks the following: if we can afford to spend the same time, by how much can we improve the solution using p processors?

We illustrate the quality up question on the cyclic 7-roots benchmark problem [BF91]. The online SymPy documentation [SymPyDocs] uses the cyclic 4-roots problem to illustrate its `nonlinsolve` method.

The function defined below returns the elapsed performance of the blackbox solver on the cyclic 7-roots benchmark problem, for a number of tasks and a precision equal to double, double double, or quad double arithmetic.

```
def qualityup(nbtasks=0, precflag='d'):
    """
    Runs the blackbox solver on a system.
    The default uses no tasks and no multiprecision.
    The elapsed performance is returned.
    """
    from phcpy.families import cyclic
    from phcpy.solver import solve
    from time import perf_counter
    c7 = cyclic(7)
    tstart = perf_counter()
    s = solve(c7, verbose=False, tasks=nbtasks, \
              precision=precflag, checkin=False)
    return perf_counter() - tstart
```

The function above is applied in an interactive Python script, prompting the user for the number of tasks and precision, This scripts runs in a Terminal window and prints the elapsed performance returned by the function. If the quality of the solutions is defined as the working precision, then to answer the quality up question, one considers how many processors are needed to compensate for the overhead of the multiprecision arithmetic.

Although cyclic 7-roots is a small system for modern computers, the cost of tracking all solution paths in double double and quad double arithmetic causes significant overhead. The script above was executed on a 2.2 GHz Intel Xeon E5-2699 processor in a CentOS Linux workstation with 256 GB RAM and the elapsed performance is in Table 1.

precision	**d**	**dd**	**qd**
elapsed perform.	5.45	42.41	604.91
overhead factor	1.00	7.41	110.99

TABLE 1: *Elapsed performance of the blackbox solver in double, double double, and quad double precision.*

Table 2 demonstrates the reduction of the overhead caused by the multiprecision arithmetic by multitasking.

tasks	**8**	**16**	**32**
dd	7.56	5.07	3.88
qd	96.08	65.82	44.35

TABLE 2: *Elapsed performance of the blackbox solver with 8, 16, and 32 path tracking tasks, in double double and quad double precision.*

Notice that the 5.07 in Table 2 is less than the 5.45 of Table 1: with 16 tasks we doubled the precision and finished

the computations in about the same time. The 42.41 and 44.35 in Table 2 are similar enough to state that with 32 instead of 1 task we doubled the precision from double double to quad double precision in about the same time.

The data in Table 2 is visualized in Fig. 3. The interpolation allows us to estimate running times for a number of tasks different from the measured run times. To answer the original quality up question, one could interpolate between the sizes of working precision to answer the following quality up question. If we can afford to spend the same time as on one path tracking task, then how many extra decimal places can we gain with p path tracking tasks?

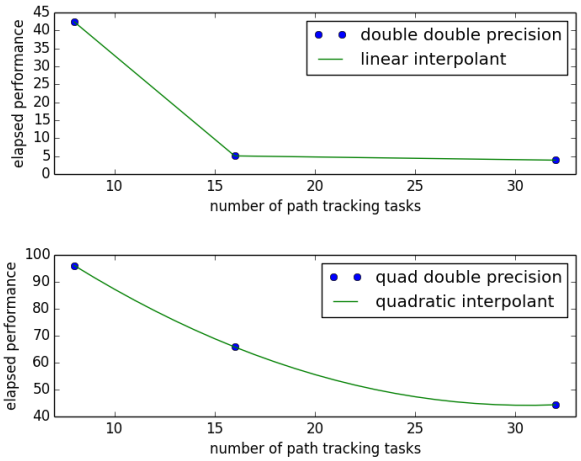


Fig. 3: *Interpolated elapsed performances.*

Precision is a crude measure of quality. Another motivation for quality up by parallelism is to compensate for the cost overhead caused by arithmetic with power series. Power series are hybrid symbolic-numeric representations for algebraic curves.

Positive Dimensional Solution Sets

Solving a system has evolved in meaning, from computing approximations of all its isolated solutions, to finding the numerical irreducible decomposition of the solution set. The numerical irreducible decomposition includes not only the isolated solutions, but also the representations for all positive dimensional solution sets. Such representations consist of sets of *generic points*, partitioned along the irreducible factors.

To illustrate this expanded sense of a 'solution', we consider the twisted cubic, known in algebraic geometry as the first non-trivial space curve. We use this example to illustrate two different representations of this space curve:

1) In a *witness set* construction, the given polynomial equations are augmented with as many generic hyperplanes as the dimension of the solution set. The solutions which satisfy the system and the augmented equations are generic points. As the degree of the twisted cubic is three, we find three points on a random plane intersecting the cubic.

```
pols = ['x*y - z;', 'x^2 - y;']
from phcpy.sets import embed
from phcpy.solver import solve
embp = embed(3, 1, pols)
sols = solve(embp, verbose=False)
print('#generic points :', len(sols))
```

   The above snippet constructs the embedding for the equations that define the twisted cubic. The solutions of this embedding represent the curve. Moving the added plane and tracking the solution paths starting at the three generic points will provide many more samples of the curve.

2) A *series expansion* for the solution starts its development at some point(s) in a coordinate hyperplane. In this hyperplane, the curve intersects the solution set at some point(s). For a simple example as the twisted cubic, the series development defines an exact solution after the initial term. Consider the snippet:

```
pols = ['x*y - z;', 'x^2 - y;']
from phcpy.maps import solve_binomials
maps = solve_binomials(3, pols, \
           puretopdim=True)
for sol in maps:
    print(sol)
```

   The output of the above snippet is

```
['x - (1+0j)*t1**1', 'y - (1+0j)*t1**2', \
 'z - (1+0j)*t1**3', 'dimension = 1', \
 'degree = 3']
```

   which corresponds to the parametric respresentation (t, t^2, t^3) of the twisted cubic.

Many interesting polynomial systems have isolated solutions and positive dimensional solution sets. We consider again the family of cyclic n-roots problems, now for $n = 8$, [BF94]. While for $n = 7$ all roots are isolated points, there is a one dimensional solution curve of cyclic 8-roots of degree 144. This curve decomposes in 16 irreducible factors, eight factors of degree 16 and eight quadratic factors, adding up to $8 \times 16 + 8 \times 2 = 144$.

Consider the following code snippet.

```
from phcpy.phcpy2c3 import py2c_set_seed
from phcpy.factor import solve
from phcpy.families import cyclic
py2c_set_seed(201905091)  # for a reproducible run
c8 = cyclic(8)
sols = solve(8, 1, c8, verbose=False)
witpols, witsols, factors = sols[1]
deg = len(witsols)
print('degree of solution set at dimension 1 :', deg)
print('number of factors : ', len(factors))
_, isosols = sols[0]
print('number of isolated solutions :', len(isosols))
```

The output of the script is

```
degree of solution set at dimension 1 : 144
number of factors :  16
number of isolated solutions : 1152
```

This numerical output is the essence of the blackbox solver for positive dimensional solution sets [Ver18].

Survey of Applications

We consider some examples from various literatures which apply polynomial constraint solving. The first two examples use phcpy in particular as a research tool. The remaining three are broader examples representing current uses of numerical algebraic geometry in other STEM fields.

Rigid Graph Theory

The conformations of proteins [LML14], molecules [EM99], and robotic mechanisms (discussed further below) can be studied by counting and classifying unique mechanisms, i.e. real embeddings

of graphs with fixed edge lengths, modulo rigid motions, per Bartzos et al. [BELT18].

Consider a graph G whose edges $e \in E_G$ each have a given length d_e. A graph embedding is a function that maps the vertices of G into D-dimensional Euclidean space (especially $D = 2$ or 3). Embeddings which are 'compatible' are those which preserve G's edge lengths. The number of unique mechanisms is thus a function of G and $\mathbf{d}$. An upper bound over all d and G with k vertices (yielding lower bounds for graphs with $n \geq k$ vertices, unless the upper bound is infinite) can be computed. In particular, the Cayley-Menger matrix of $\mathbf{d}$ [LLMM14] (i.e., the squared distance matrix with a row and column of 1s prepended, except that its main diagonal is 0s) is an algebraic system, proportional to the mixed volume. Certain of its square subsystems characterize the mechanism in terms of these bounds on unique mechanisms.

Bartzos et al. implemented, using `phcpy`, a constructive method yielding all 7-vertex minimally rigid graphs in 3D space (the smallest open case) and certain 8-vertex cases previously uncounted. A graph G is generically rigid if, for any given edge lengths d, none of its compatible embeddings (into a generic configuration such that vertices are algebraically independent) are continuously deformable. G is minimally rigid if removing any one of its edges yields a non-rigid mechanism.

`phcpy` was used to find edge lengths with maximally many real embeddings, exploiting the flexibility of being able to specify their starting system. This sped up their algorithm by perturbing the solutions of previous systems to find new ones.

Many iterations of sampling have to be performed if the wrong number of real embeddings is found; in each case, a different subgraph is selected based on a heuristic implemented by the `DBSCAN` class of `scikit-learn` (illustrating the value of a scientific Python ecosystem). The actual number of real embeddings is known from an enumeration of unique graphs constructed by Henneberg steps in, for instance, SageMath.

Model Selection & Parameter Inference

It is often useful to know all the steady states of a biological network, as represented by a nonlinear system of ordinary differential equations, with some conserved quantities. These two lists of polynomials (from rates of change of form $\dot{x} = p(x)$, by letting $\dot{x} = 0$; and from conservation laws of form $c = \sum x_i$ by subtracting c from both sides) have a zero set which is a steady-state variety, that can be explored numerically via polynomial homotopy continuation.

Parameter homotopies were used by Gross et al. [GHR16] to perform model selection on a mammalian phosphorylation pathway, determining whether the kinase acts processively (i.e. adding more than one phosphate at once, which it does not in vitro). Their analysis validated experimental work showing processivity in vivo. In doing so, they obtained >50x speedup over non-parameter homotopies (for running times in minutes, not hours) on systems tracking 20 paths.

Critical Point Computation

Polynomial homotopy continuation has also been adapted to the field of chemical engineering to locate critical points of multicomponent mixtures [SWM16], i.e., temperature and pressure satisfying a multi-phase equilibrium.

A remarkable variety of systems of constraint also take on polynomial form, or can be approximated thereby, in various sciences. Diverse problems in the analysis of belief propagation (in graphical models) [KMC18], hyperbolic conservation laws (in PDEs) [HHS13], and vacuum moduli spaces (in supersymmetric field theory) [HHM13] have been addressed using polynomial homotopy continuation.

Algebraic Kinematics

We have discussed an application of numerical methods to counting unique instances of rigid-body mechanisms. In fact, kinematics and numerical algebraic geometry have a close historical relationship. Following Wampler and Sommese [WS11], other geometric problems arising from robotics include **analysis** of specific mechanisms e.g.,:

- Motion decomposition - into assembly modes (of individual mechanisms) or subfamilies of mechanisms (with varying mobility);
- Mobility analysis - degrees of freedom of a mechanism (sometimes exceptional), sometimes specific to certain configurations (e.g., gimbal lock);
- Kinematics - effector position given parameters (forward kinematics), and vice versa (inverse kinematics, e.g. used in computer animation);
- Singularity analysis - detection of situations where the mechanism can move without change to its parameters (input singularity), or the parameters can change without movement of the mechanism (output singularity);
- Workspace analysis - determining all possible outputs of the mechanism, i.e.: reachable poses;

...as well as the **synthesis** of mechanisms that can reach certain sets of outputs, or that can be controlled by a certain input/output relationship.

Fig. 4 illustrates a reproduction of one synthesis result in the mechanism design literature [MW90]. Given five points, the problem is to determine the length of two bars so their coupler curve passes through the five given points.

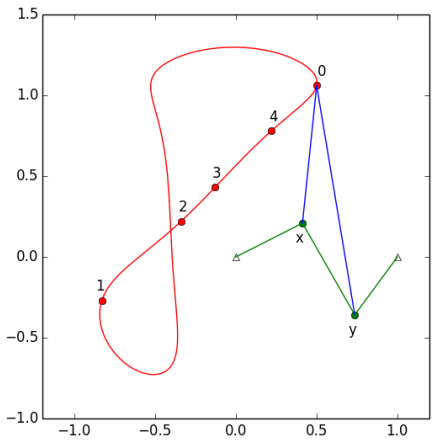


Fig. 4: *The design of a 4-bar mechanism.*

This example is part of the tutorial of phcpy and the scripts to reproduce the results are in its source code distribution. The equations are generated with sympy [SymPy] and the plots are made with matplotlib [Hun07].

Continuation homotopies were developed as a substitute for algebraic elimination that was more robust to special cases, yet

still tractable to numerical techniques. Research in kinematics increasingly relies on such algorithms [WS11].

Systems Biology

Whether a model biological system is multistationary or oscillatory, and whether this depends on its rate constants, are all properties of its steady-state locus. Following the survey of Gross et al. [GBH16] regarding uses of numerical algebraic geometry in this domain, one might seek to:

- determine which values of the rate and conserved-quantity parameters allow the model to have multiple steady states;
- evaluate models with partial data (subsets of the x_i) and reject those which don't agree with the data at steady state;
- describe all the states accessible from a given state of the model, i.e. that state's stoichiometric compatibility class (or basin of attraction);
- determine whether rate parameters of the given model are identifiable from concentration measurements, or at least constrained.

For large real-world models in systems biology, these questions of algebraic geometry are only tractable to numerical methods scaling to many dozens of simultaneous equations.

Conclusion

From these examples, we see that polynomial homotopy continuation has wide applicability to STEM fields. Moreover, phcpy is an accessible interface to the technique, capable of high performance whilst producing certifiable and reproducible results.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1440534.

References

[BHSW13] D. J. Bates, J. D. Hauenstein, A. J. Sommese, and C. W. Wampler. *Numerically solving polynomial systems with Bertini*, volume 25 of Software, Environments, and Tools, SIAM, 2013.

[BELT18] E. Bartzos, I. Z. Emiris, J. Legersky, and E. Tsigaridas. *On the maximal number of real embeddings of spatial minimally rigid graphs*. In the Proceedings of the 2018 International Symposium on Symbolic and Algebraic Computation (ISSAC 2018), pages 55-62, ACM 2018. DOI 10.1145/3208976.3208994.

[Bertini2.0] Bertini 2.0: The redevelopment of Bertini in C++. https://github.com/bertiniteam/b2

[BF91] J. Backelin and R. Fröberg. *How we proved that there are exactly 924 cyclic 7-roots.* In the Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation (ISSAC'91), pages 103-111, ACM, 1991. DOI 10.1145/120694.120708.

[BF94] G. Björck and R. Fröberg. *Methods to "divide out" certain solutions from systems of algebraic equations, applied to find all cyclic 8-roots.* In Analysis, Algebra and Computers in Mathematical Research, Proceedings of the twenty-first Nordic congress of mathematicians, edited by M. Gyllenberg and L. E. Persson, volume 564 of Lecture Notes in Pure and Applied Mathematics, pages 57-70. Dekker, 1994.

[BSVY15] N. Bliss, J. Sommars, J. Verschelde, X. Yu. *Solving polynomial systems in the cloud with polynomial homotopy continuation.* In the Proceedings of the 17th International Workshop on Computer Algebra in Scientific Computing (CASC 2015), edited by V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, volume 9301 of Lecture Notes in Computer Science, pages 87-100, Springer-Verlag, 2015. DOI 10.1007/978-3-319-24021-3_7.

[D3] M. Bostock, V. Ogievetsky, and J. Heer *D3 Data-Driven Documents.* IEEE Transactions on Visualization and Computer Graphics, 17, pages 2301–2309, 2011. DOI 10.1109/TVCG.2011.185.

[BT18] P. Breiding and S. Timme. *HomotopyContinuation.jl: A package for homotopy continuation in Julia.* In the proceedings of ICMS 2018, the 6th International Conference on Mathematical Software, South Bend, IN, USA, July 24-27, 2018, edited by J. H. Davenport, M. Kauers, G. Labahn, and J. Urban, volume 10931 of Lecture Notes in Computer Science, pages 458-465. Springer-Verlag, 2018. DOI 10.1007/978-3-319-96418-8.

[Chu06] W. J. Chun. *Core Python Programming.* Prentice Hall, 2nd Edition, 2006.

[CD18] M. Culler and N. M. Dunfield. *Orderability and Dehn filling.* Geometry and Topology, 22: 1405-1457, 2018. DOI 10.2140/gt.2018.22.1405.

[EM99] I.Z. Emiris and B. Mourrain. *Computer algebra methods for studying and computing molecular conformations.* Algorithmica 25, pages 372–402, 1999. DOI: 10.1007/PL00008283.

[APP] *explorable circle tangency* https://github.com/JazzTap/mcs563/tree/master/Apollonius]

[HHM13] J. Hauenstein, Y.-H. He, and D. Mehta. *Numerical elimination and moduli space of vacua.* Journal of High Energy Physics, 83. 2013. DOI: 10.1007/JHEP09(2013)083.

[HHS13] W. Hao, J. D. Hauenstein, C.-W. Shu, A. J. Sommese, Z. Xu, and Y.-T. Zhang. *A homotopy method based on WENO schemes for solving steady state problems of hyperbolic conservation laws.* Journal of Computational Physics, 250, pages 332–346. 2013. DOI: 10.1016/j.jcp.2013.05.008.

[HLB01] Y. Hida, X. S. Li, and D. H. Bailey. *Algorithms for quad-double precision floating point arithmetic.* In the Proceedings of the 15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), pages 155--162. IEEE Computer Society, 2001. DOI 10.1109/ARITH.2001.930115.

[HCJL] A Julia package for solving systems of polynomials via homotopy continuation. https://github.com/JuliaHomotopyContinuation

[Hun07] J. D. Hunter. *Matplotlib: A 2D Graphics Environment.* Computing in Science and Engineering 9(3): 90-95, 2007. DOI 10.1109/MCSE.2007.55.

[GLW05] T. Gao, T.Y. Li, and M. Wu. *Algorithm 846: MixedVol: a software package for mixed-volume computation.* ACM Trans. Math. Softw., 31(4):555-560, 2005. DOI 10.1145/1114268.1114274.

[GBH16] E. Gross, D. Brent, K. L. Ho, D. J. Bates, and H. A. Harrington. *Numerical algebraic geometry for model selection and its application to the life sciences.* Journal of The Royal Society Interface, 13: 20160256. 2016. DOI: 10.1098/rsif.2016.0256.

[GHR16] E. Gross, H. A. Harrington, Z. Rosen, and B. Sturmfels. *Algebraic Systems Biology: A Case Study for the Wnt Pathway.* Bulletin of Mathematical Biology 78, pages 21–51, 2016. DOI: 10.1007/s11538-015-0125-1.

[GPV13] E. Gross, S. Petrović, and J. Verschelde. *Interfacing with PHCpack.* The Journal of Software for Algebra and Geometry: Macaulay2, 5:20-25, 2013. DOI 10.2140/jsag.2013.5.20.

[HS95] B. Huber and B. Sturmfels. *A polyhedral method for solving sparse polynomial systems.* Mathematics of Computation, 64(212):1541-1555, 1995. DOI 10.1090/S0025-5718-1995-1297471-4.

[HSS98] B. Huber, F. Sottile, and B. Sturmfels. *Numerical Schubert calculus.* Journal of Symbolic Computation, 26(6):767-788, 1998. DOI 10.1006/jsco.1998.0239.

[IPYW] *ipywidgets: Interactive HTML Widgets* https://github.com/jupyter-widgets/ipywidgets

[SymPy] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger. *Open source computer algebra systems: SymPy.* ACM Communications in Computer Algebra 45(4): 225-234 , 2011. DOI 10.1145/2110170.2110185.

[Pascal] *JupyterHub deployment of phcpy.* Website, accessed May 2019. 2017. https://phcpack.org

[JUPH] *JupyterHub 0.7.2 documentation* https://jupyterhub.readthedocs.io/en/0.7.2/index.html

[JUP15] *Jupyter notebook snippets menu - jupyter-contrib-nbextensions 0.5.0* https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/snippets_menu/readme.html.

[Klu16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team. *Jupyter Notebooks -- a publishing format for reproducible computational workflows*. In Positioning and Power in Academic Publishing: Players, Agents, and Agendas, edited by F. Loizides and B. Schmidt, pages 87-90. IOS Press, 2016. DOI 10.3233/978-1-61499-649-1-87.

[KMC18] C. Knoll, D. Mehta, T. Chen, and F. Pernkopf. *Fixed Points of Belief Propagation—An Analysis via Polynomial Homotopy Continuation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 40, pages 2124–2136, 2018. DOI 10.1109/TPAMI.2017.2749575.

[Ley11] A. Leykin. *Numerical algebraic geometry.* The Journal of Software for Algebra and Geometry: Macaulay2, 3:5-10, 2011. DOI 10.2140/jsag.2011.3.5.

[LVZ06] A. Leykin, J. Verschelde, and A. Zhao. *Newton's method with deflation for isolated singularities of polynomial systems.* Theoretical Computer Science, 359(1-3):111-122, 2006. DOI 10.1016/j.tcs.2006.02.018.

[LLMM14] L. Liberti, C. Lavor, N. Maculan, and A. Mucherino. *Euclidean Distance Geometry and Applications.* SIAM Review 56, no. 1 (January 2014): 3–69. DOI 10.1137/120875909

[LML14] L. Liberti, B. Masson, J. Lee, C. Lavor, and A. Mucherino. *On the number of realizations of certain henneberg graphs arising in protein conformation.* Discrete Applied Mathematics, 165, page 213–232, 2014. DOI: 10.1016/j.dam.2013.01.020.

[M2] D. R. Grayson and M. E. Stillman. Macaulay2, a software system for research in algebraic geometry. http://www.math.uiuc.edu/Macaulay2

[MT08] T. Mizutani and A. Takeda. *DEMiCs: A software package for computing the mixed volume via dynamic enumeration of all mixed cells.* In Software for Algebraic Geometry, edited by M. E. Stillman, N. Takayama, and J. Verschelde, volume 148 of The IMA Volumes in Mathematics and its Applications, pages 59-79. Springer-Verlag, 2008. DOI 10.1007/978-0-387-78133-4.

[MW90] A. P. Morgan and C. W. Wampler. *Solving a Planar Four-Bar Design Using Continuation.* Journal of Mechanical Design, 112(4): 544-550, 1990. DOI 10.1115/1.2912644.

[NAG4M2] *Branch NAG of M2 repository.* https://github.com/antonleykin/M2/tree/NAG

[MSDB] *MySQLdb 1.2.4b4 documentation* https://mysqlclient.readthedocs.io/

[PHCPY] *phcpy 0.9.5 documentation* http://homepages.math.uic.edu/~jan/phcpy_doc_html/

[Sage] The Sage Developers. *SageMath, the Sage Mathematics Software System, Version 7.6*. https://www.sagemath.org, 2016. DOI 10.5281/zenodo.820864.

[SJ05] W. Stein and D. Joyner. *Sage: System for algebra and geometry experimentation.* ACM SIGSAM Bulletin 39(2): 61-64, 2005. DOI 10.1145/1101884.1101889.

[SWM16] H. Sidky, J. K. Whitmer, and D. Mehta. *Reliable mixture critical point computation using polynomial homotopy continuation.* AIChE Journal. Thermodynamics and Molecular-Scale Phenomena, 62(12): 4497-4507, 2016. DOI 10.1002/aic.15319.

[SVW03] A. J. Sommese, J. Verschelde, and C. W. Wampler. *Numerical irreducible decomposition using PHCpack.* In Algebra, Geometry and Software Systems, edited by M. Joswig and N. Takayama, pages 109-130, Springer-Verlag 2003. DOI 10.1007/978-3-662-05148-1_6.

[SVW05] A. J. Sommese, J. Verschelde, and C. W. Wampler. *Introduction to numerical algebraic geometry.* In Solving Polynomial Equations, Foundations, Algorithms, and Applications, edited by A. Dickenstein and I. Z. Emiris, pages 301-337, Springer-Verlag 2005. DOI 10.1007/3-540-27357-3_8.

[SymPyDocs] *SymPy 1.3 documentation.* https://docs.sympy.org/latest/index.html

[Ver99] J. Verschelde. *Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation*, ACM Trans. Math. Softw., 25(2):251-276, 1999. DOI 10.1145/317275.317286.

[Ver14] J. Verschelde. *Modernizing PHCpack through phcpy.* Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), edited by P. de Buyl and N. Varoquaux, pages 71-76, 2014.

[Ver18] J. Verschelde. *A Blackbox Polynomial System Solver for Shared Memory Parallel Computers.* In Computer Algebra in Scientific Computing, 20th International Workshop, CASC 2018, Lille, France, edited by V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, volume 11077 of Lecture Notes in Computer Science, pages 361-375. Springer-Verlag, 2018. DOI 10.1007/978-3-319-99639-4_25.

[VC93] J. Verschelde and R. Cools. *Symbolic homotopy construction.* Applicable Algebra in Engineering, Communication and Computing, 4(3):169-183, 1993. DOI 10.1007/BF01202036.

[VVC94] J. Verschelde, P. Verlinden, and R. Cools. *Homotopies exploiting Newton polytopes for solving sparse polynomial systems.* SIAM Journal on Numerical Analysis 31(3):915-930, 1994. DOI 10.1137/0731049.

[VY15] J. Verschelde and X. Yu. *Polynomial Homotopy Continuation on GPUs.* ACM Communications in Computer Algebra, volume 49, issue 4, pages 130-133, 2015. DOI 10.1145/2893803.2893810.

[WS11] C. W. Wampler & A. J. Sommese *Numerical algebraic geometry and algebraic kinematics.* Acta Numerica, 20, pages 469–567. 2011. DOI: 10.1017/S0962492911000067.

[Yu15] X. Yu. *Accelerating Polynomial Homotopy Continuation on Graphics Processing Units.* PhD thesis, University of Illinois at Chicago, 2015.

Optimizing Python-Based Spectroscopic Data Processing on NERSC Supercomputers

Laurie A. Stephey[‡*], Rollin C. Thomas[‡], Stephen J. Bailey[§]

Abstract—We present a case study of optimizing a Python-based cosmology data processing pipeline designed to run in parallel on thousands of cores using supercomputers at the National Energy Research Scientific Computing Center (NERSC).

The goal of the Dark Energy Spectroscopic Instrument (DESI) experiment is to better understand dark energy by making the most detailed 3D map of the universe to date. Over a five-year period starting this year (2019), around 1000 CCD frames per night (30 per exposure) will be read out from the instrument and transferred to NERSC for processing and analysis on the Cori and Perlmutter supercomputers in near-real time. This fast turnaround helps DESI monitor survey progress and update the next night's observing schedule.

The DESI spectroscopic pipeline for processing these data is written almost exclusively in Python. Using Python allows DESI scientists to write very readable and maintainable scientific code in a relatively short amount of time, which is important due to limited DESI developer resources. However, the drawback is that Python can be substantially slower than more traditional high performance computing languages like C, C++, and Fortran.

The goal of this work is to improve the performance of DESI's spectroscopic data processing pipeline at NERSC while satisfying their productivity requirement that the software remain in Python. Within this space we have obtained specific (per node-hour) throughput improvements of over 5x and 6x on the Cori Haswell and Knights Landing partitions, respectively. Several profiling techniques were used to determine potential areas for improvement including Python's cProfile and line_profiler packages, and other tools like Intel VTune and Tau. Once we identified expensive kernels, we used the following techniques: 1) JIT-compiling hotspots using Numba and 2) restructuring the code to lessen the impact of calling expensive functions. Additionally, we seriously considered substituting MPI parallelism for Dask, a more flexible and robust alternative, but have found that once a code has been designed with MPI in mind, it is non-trivial to transition it to another kind of parallelism. We will also show initial considerations for transitioning DESI spectroscopic extraction to GPUs (coming in the next NERSC system, Perlmutter, in 2020).

Index Terms—NumPy, SciPy, Numba, JIT compile, spectroscopy, HPC, MPI, Dask

Introduction

DESI is the Dark Energy Spectroscopic Instrument [noae]. Though dark energy is estimated to comprise over 70 percent of our universe, it is not currently well-understood [PR03], [MWW13]. Many experiments, including DESI, are seeking to uncover more information about the nature of dark energy. The goal of the DESI experiment is, over 5 years, to map 30 million galaxies and use spectroscopically obtained redshift data to measure their distances. The statistical properties of this 3D galaxy map will help shed light on the physical nature of dark energy and its role in the evolution of the universe. An image of the Mayall telescope, on Kitt Peak, Arizona, where the DESI instrument is installed, is shown in Figure 1.

Fig. 1: *A photograph of the Mayall telescope (large dome in the center of the image), where the DESI instrument has been installed, on Kitt Peak, Arizona.*

In fall 2019 DESI will begin sending batches of CCD images nightly to the National Energy Research Scientific Computing Center (NERSC) for data processing. Each exposure contains the data from 5000 galaxies, quasars, stars, and reference calibrators, routed by fiber optic cables from the telescope to 10 spectrographs with 3 CCDs (red, blue, and infrared) and 500 spectra each. This means that each exposure contains 30 individual images (with each exposure totaling about 6 GB). DESI expects to collect over 30 exposures in a typical night, resulting in over 1000 images.

A small subset of example data are shown in Figure 2 with 21 spectra distributed horizontally and different wavelengths of light dispersed vertically. This image represents less than one millionth of the DESI data obtained per night. Most spectra look the same since all fibers see the same night sky. The slight excess in the middle of the leftmost fiber is the signal from a distant galaxy. Even though this is faint compared to the sky background, this example is in the brightest 15% of galaxies that DESI will observe.

Compared to prior galaxy redshift surveys, DESI will observe fainter, more distant objects at lower signal-to-noise, necessitating more sophisticated algorithms to optimally extract the signal from the data. This requires a full 2D modeling of the data,

* *Corresponding author: lastephey@lbl.gov*
‡ *NERSC*
§ *LBL*

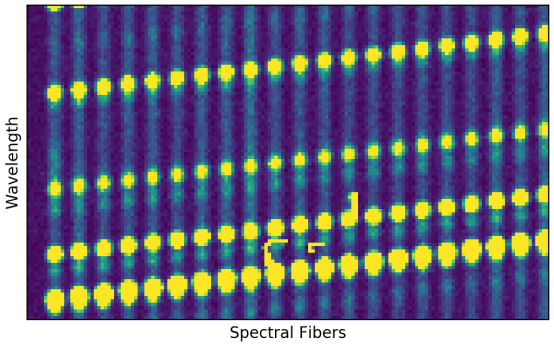


Fig. 2: *Example DESI data showing spectra from 21 of the 5000 fibers distributed horizontally, with wavelengths dispersed vertically. Most spectra look the same since they all see the same sky background light. The slight excess of light in the middle of the leftmost spectrum is the signal from a distant galaxy.*

fitting multiple spectra and wavelengths simultaneously using the "spectroperfectionism" algorithm [BS10], which is only computationally feasible due to a divide-and-conquer technique. This case study focuses on this spectral extraction part of the data processing pipeline since it is the algorithmically most expensive step; it includes eigenvalue decomposition, special function evaluation, and all the necessary bookkeeping required to manage the spectral data in each exposure.

The overarching goal of this work is to speed up the DESI experiment's Python spectroscopic data processing pipeline on the Cori supercomputer's KNL partition at NERSC. NERSC [noag] is the largest Department of Energy computing facility in terms of number of users (7000) and scientific output [noal]. Cori is NERSC's current flagship supercomputer, a Cray XC40 with a theoretical peak performance of 28 PF, comprised of approximately 20 percent Intel Haswell nodes and 80 percent manycore Intel Knights Landing (KNL) nodes.

Achieving good performance with the manycore KNL nodes has proven difficult for many science teams. Because the Haswell nodes are "easier" to use (i.e. applications often run faster on them out of the box), they are increasingly crowded. For this reason NERSC established a program called NESAP (NERSC Exascale Science Applications Program, [noah]) to help science teams transition successfully to the KNL nodes. NESAP provides technical expertise from NERSC staff and vendors like Intel and Cray to science teams to improve the performance of their application on the Cori KNL partition and prepare for the manycore future of high-performance computing (HPC). NESAP's goal is to help move a large fraction of the NERSC workload from the Haswell to the KNL partition; this will ease queue wait times and help increase job throughput for all users.

Achieving optimal Python performance on KNL is especially challenging due its slower clock speed and difficulty taking advantage of the KNL AVX-512 vector units (which is not possible in native Python). A more detailed discussion of the difficulties of extracting Python performance on KNL can be found in [RTD+17]. This case study is borne out of DESI's participation in the NERSC NESAP program.

Despite these difficulties, DESI requested that their code should not be re-written in another language like C due to their own limited developer resources. They did consider both Cython [noad] and Numba [noai] as options for improving performance, but after some initial testing they found that both delivered approximately equivalent speedups for their specific test cases. Citing Numba's ease of use, automatic compilation, and ability to gracefully fall back to non-compiled code, they requested that NESAP proceed with Numba-based optimizations where necessary.

In what follows we will present a case study that describes how a Python image processing pipeline was optimized *without rewriting the code in another language like C* for increased throughput of 5-7x on a high-performance system. We will describe our workflow of using profiling tools to find candidate kernels for optimization and we will describe how we used just in time compiling to speed up these kernels. We will also describe our efforts to restructure the code to minimize the impact of calling expensive kernels. We will compare parallelization strategies using MPI and Dask, and finally, we will discuss a preliminary study for moving the DESI code to GPUs.

Profiling the Code

Our first step in this study was to use profiling tools to determine places in the DESI code where it was worthwhile to target our optimization efforts. We made heavy use of tools designed especially for Python. In general our process was to start with the simplest tools and then, when we knew what we were looking for, use the more complex tools.

We should note that we profiled the DESI code on both Cori Haswell and KNL nodes. There were some minor differences in the relative time spent in each kernel between the two architectures, but overall the same patterns were present on both Haswell and KNL.

cProfile

Python's built-in cProfile package [noaa] was the first tool we used for collecting profiling data. We found cProfile simple and quick to use because it didn't require any additions or changes to the DESI code. cProfile can write data to a human-readable file, but we found that using either Snakeviz [noaq] or gprof2dot [Fon19] to visualize the profiling data was substantially more clear and useful.

An example of data collected using cProfile and visualized with gprof2dot is shown in Figure 3. We prefer gprof2dot to Snakeviz visualizations because they are static images instead of browser-based. This makes them easier to store, share, quickly view, and embed in papers and talks. If you prefer accessing the cProfile data interactively, and clicking on a function to see all of its children, for example, Snakeviz can provide this functionality. However, we found the several extra steps required to use Snakeviz, and the difficulty storing and sharing the visualizations, made it less appealing than gprof2dot.

Examining the visualized cProfile data allowed us to identify expensive kernels in the DESI calculation. In Figure 3, the functions are color-coded according to how much total time is spent in each of them. In this example, the function `traceset` accounts for approximately 37 percent of the total runtime and was a good candidate for optimization efforts.

Information like that shown in Figure 3 is nevertheless incomplete in that it can only provide detail at the function level. From these data alone it was difficult to know what specifically in the

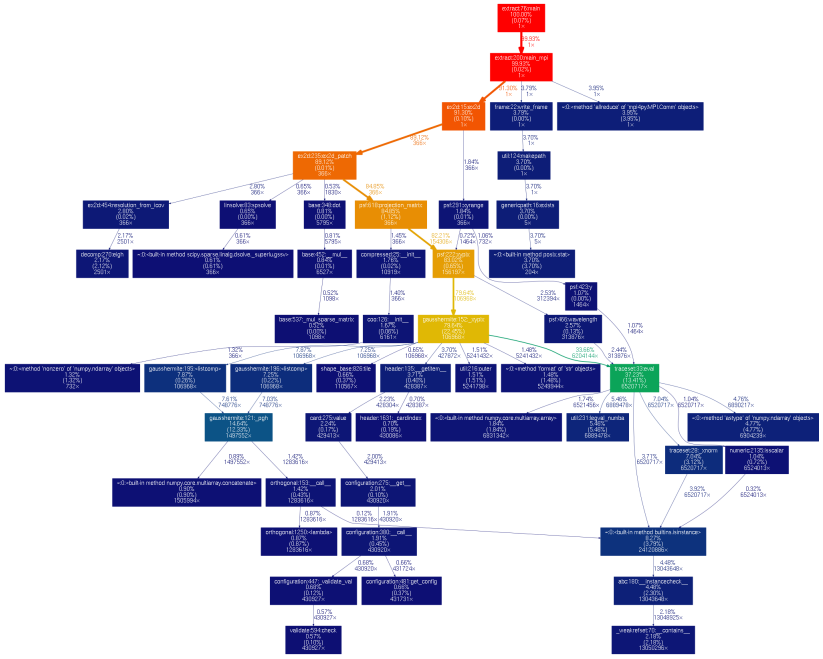


Fig. 3: *This is an example image created from data collected using cProfile and visualized using gprof2dot [Fon19]. This profile was obtained from an early stage in the NESAP optimization effort.*

function "traceset" was so time-consuming. Once we had a list of expensive kernels from our cProfile/gprof2dot analysis, we started using the line_profiler tool.

line_profiler

line_profiler [Ker19] is an extremely useful tool which provides line-by-line profiling information for a Python function. However, this more detailed information comes at a cost: the user must manually decorate functions that he or she wishes to profile. For a small code this exercise might be trivial, but for the many thousand line DESI code 1) hand-decorating every function would have been extremely time-consuming and 2) searching through the line_profiler output data to find expensive functions would have also been cumbersome and potentially error-prone. For this reason we recommend starting with cProfile and then moving to line_profiler once the user has identified a few key functions of interest.

Once decorated, line_profiler provides a great deal of information for each line of the function, including how many times each line was invoked and the total amount of time spent on each line. An example of line_profiler output for the function `xypix` is shown in Figure 4. This information was vital to our optimization efforts because it could point to functions that were particularly expensive, such as numpy's `legval` or scipy's `erf`. Once we had this information, we could make decisions about how to reduce the time spent in these functions, either by speeding up the functions themselves through JIT compiling, or by restructuring the code to make the functions either less expensive or avoid calling them as often. We will describe these approaches in the sections that follow.

Together, cProfile and line_profiler were sufficient for almost all of the performance optimization work in this case study. However, because the DESI extraction code is an MPI code, these profiling tools do have some limitations. Both of these tools can be used to collect data for each MPI rank, but visualizing and using

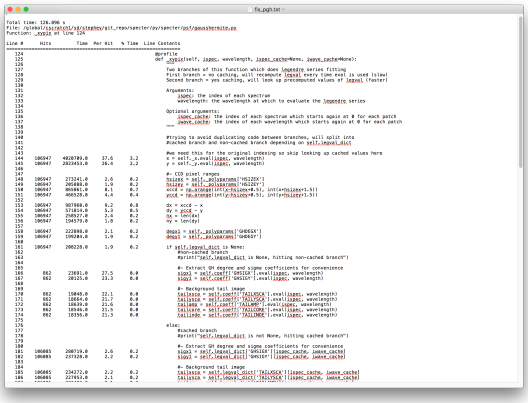


Fig. 4: *Here is a sample output window from line_profiler [Ker19] for the function "xypix". The clear, human-readable output files produced by line_profiler are a very nice feature.*

the information in a meaningful way is challenging, especially when there are 68 outputs from a KNL chip, for example.

VTune and Tau

Once we reached the point where we wanted to investigate 1) each individual MPI rank and 2) whether all ranks were appropriately load-balanced, we needed more powerful profiling tools like Intel VTune [adm] and Tau [noar]. While VTune is a very powerful general tool for studying code, we found that it was difficult to get the information we wanted in a clear, understandable format. For example, VTune would often display extremely low-level information that obfuscated the higher-level Python calls we were trying to investigate. We found gprof2dot and Snakeviz visualizations easier to navigate than the VTune GUI. We ultimately found the Tau profiler more useful and well-suited for our application, although we should note that we required the help of the Tau

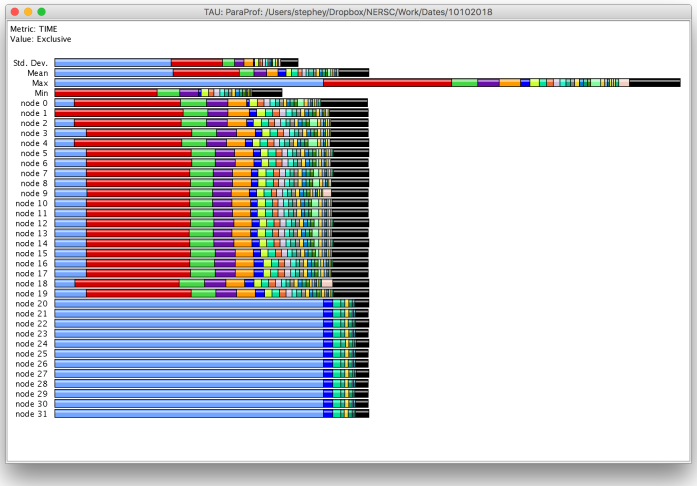


Fig. 5: *A sample Tau [noar] output for the DESI spectral extraction code on a Haswell processor (which has 32 ranks). It is clear from this output that only 20 of the ranks are being utilized. This motivated the restructure to allow parallelization of subbundles, rather than bundles, which could more flexibly utilize the whole processor's resources.*

developers to build it. (Tau works best when it is built for the type of application you will profile. In our case it was a Python MPI code running on a Cray system, all of which are configurations that Tau supports.) Though building a profiling tool from scratch was non-trivial, it was also very possible with the help of the Tau team. Once built, Tau provided clear information about how each MPI rank was occupied and how each rank compared to the others. A sample Tau output window is shown in Figure 5. These profiling data were obtained while the DESI frame was parallelized over bundles which left 12 of the 32 Haswell ranks unoccupied. It is clear from this Tau visualization that we were not making good use of processor resources.

Just-in-time (JIT) Compilation with Numba

The first major approach to achieve speedups in this work has been to focus on making expensive functions run more quickly. To achieve this, we have used Numba [LPS15], a just-in-time compiler for Python.

We used Numba for three functions that, through profiling, we identified as expensive. These functions were 1) `numpy.polynomial.legendre.legval` [noaj], 2) `scipy.special.erf` [noao], and 3) `scipy.special.hermitenorm` [noap], which henceforth we will refer to as `legval`, `erf`, and `hermitenorm`.

`legval` was perhaps the most straightforward of these three to JIT compile. Unlike Python, Numba requires that all variables and arrays cannot change type, nor can they change size (e.g. this information must be known prior at compile time). This necessitated several small changes to the `legval` algorithm to put it in the form required by Numba. Several other lines of the function that performed type checking were removed. This placed the onus on the developer to make sure the correct types are supplied, which was acceptable for us. The original and modified `legval` functions are shown in Figure 6.

The two scipy functions were also somewhat challenging to implement in Numba. At the time of this writing, Numba does not yet support directly compiling scipy functions. This meant that we needed to extract the core part of these scipy functions and mold them into a form that Numba would accept. For scipy `erf`, this meant translating the Fortran source code into Python. For scipy `hermitenorm`, which was fortunately already in Python, algorithmic changes similar to those we made in `legval` were necessary to ensure all variables were a constant type and size.

We should note that we tried to cache the compiled Numba functions with the `cache=True` option to save time, but with larger numbers of MPI ranks, we found that this sometimes caused a data race between the Numba caches written by each rank. To avoid this problem we considered using ahead of time (AOT) instead of JIT compiling but since implementing this change was somewhat awkward, for now we have removed the `cache=True` setting and will consider using AOT in the future.

Restructuring the Code

Restructuring the code was the second major optimization strategy we used. In the three subsections that follow, we will describe three types of restructuring efforts that we have completed or will soon complete. In the first restructure, we have altered the code to process smaller matrices at a time to reduce the performance hit we take in the `scipy.linalg.eigh` function. In the second restructure, we have changed the code to avoid calling an expensive function, `numpy.polynomial.legendre.legval`. In the third restructure, which is currently in progress, we are changing the structure of parallelism to divide the problem by subbundle rather than by bundle. This restructure doesn't itself provide a performance boost, but it does provide substantially increased flexibility for the DESI code.

Implement Subbundles

Profiling data indicated that when matrix sizes were large, `scipy.linalg.eigh`, a key part of the spectroperfectionism extraction, was extremely slow. This is not surprising because Jacobi eigenvalue algorithms scale as $O(n^3)$ [PTV+92]. One recommendation from an Intel Dungeon session (a collaborative hack session between NESAP teams and Intel engineers) was to reduce the number of fibers processed at a time. This meant dividing a single bundle of 25 fibers into 6 smaller groups known as subbundles. By computing the eigenvalues of more, but smaller, covariance matrices, DESI was able to reduce their computation time. It is important to mention that DESI can only use this type of approach because they have been careful to design their experiment so as to minimize crosstalk between individual fibers, which results in a sparse covariance matrix. We will also note that there was nothing magical about the number 6; anywhere from 2 to 10 subbundles provided a similar performance increase on both KNL and Haswell. While this strategy was successful on CPUs, we will revisit this strategy in the section "Does it Make Sense to Run DESI Code on GPUs".

Add Cached `legval` Values

Another outcome from the Intel Dungeon session was the recommendation to restructure the code to avoid calling `legval`. The problem with `legval` wasn't just that it was an expensive function; rather, it was also contributing to a large fraction of the total runtime because it was called millions of times for each CCD image in the DESI spectral extraction calculation. Worse, `legval` was called with scalar values even though it was able to handle vector inputs.

(A)

```
def legval(x, c, tensor=True):
    c = np.array(c, ndmin=1, copy=0)
    if c.dtype.char in '?bBhHiIlLqQpP':
        c = c.astype(np.double)
    if isinstance(x, (tuple, list)):
        x = np.asarray(x)
    if isinstance(x, np.ndarray) and tensor:
        c = c.reshape(c.shape + (1,)*x.ndim)

    if len(c) == 1:
        c0 = c[0]
        c1 = 0
    elif len(c) == 2:
        c0 = c[0]
        c1 = c[1]
    else:
        nd = len(c)
        c0 = c[-2]
        c1 = c[-1]
        for i in range(3, len(c) + 1):
            tmp = c0
            nd = nd - 1
            c0 = c[-i] - (c1*(nd - 1))/nd
            c1 = tmp + (c1*x*(2*nd - 1))/nd
    return c0 + c1*x
```

(B)

```
import numba
@numba.jit(nopython=True,cache=False)
def legval_numba(x, c):
    nd=len(c)
    ndd=nd
    xlen = x.size
    c0=c[-2]*np.ones(xlen)
    c1=c[-1]*np.ones(xlen)
    for i in range(3, ndd + 1):
        tmp = c0
        nd = nd - 1
        nd_inv = 1/nd
        c0 = c[-i] - (c1*(nd - 1))*nd_inv
        c1 = tmp + (c1*x*(2*nd - 1))*nd_inv
    return c0 + c1*x
```

Fig. 6: *(A) The official* `numpy.polynomial.legendre.legval` *function. Profiling data indicated that this was an expensive function. To conserve space the docstring has been removed. (B) Our modified* `legval` *function that was much faster than its original numpy counterpart. Note the removal of the type checking and the addition of the* `np.ones` *array to instruct Numba about the sizes of each array (and prevent them from changing during every iteration.)*

This restructuring required us to modify several major functions and redefine some of the bookkeeping that keeps track of which data corresponds to which part of the image on the CCD. Prior to the restructure, profiling data indicated that `legval` was called approximately 7 million times per frame with scalar values.

The code was restructured so that `legval` was now called 800,000 times per frame. Of course this is still a large number, but it is almost an order of magnitude fewer times than the original implementation. The calculated values were stored as key-value pairs in a dictionary. We then modified the part of the code that previously calculated `legval` to instead look up the required values stored in the dictionary.

Parallelize over Subbundles Instead of Bundles

Desipte these optimizations, the DESI code still has several known issues: poor load-balancing and rigid requirements for job sizes (9 nodes for KNL and 19 Nodes for Haswell, for example). We are in the process of addressing these issues and thought that our efforts were worth mentioning.

The goal of parallelizing over subbundles, rather than bundles, is to restructure the code to divide the spectral extraction into smaller, more flexible pieces. This will relax the previous requirement that each frame be divided into 20 bundles, which is an awkward number for NERSC hardware (and a restrictive condition in general). When completed, the 500 spectra will be more evenly doled out to 32 processors (about 16 spectra each) or 68 processors (about 7 spectra each). This means that all processors can be used for any given job size, not just for a carefully chosen job size. However, like the other restructuring efforts, we have found that implementing this change is nontrivial.

Additionally, this refactor will help improve load balancing. Since the processing time differs for the three types of DESI frames (blue, red, and infrared), prior to the refactor, the processors assigned to the blue frames finished before the infrared frames, wasting both valuable processor resources and time. In this new design, frame types will be grouped together so processor time is not wasted.

Optimization Results

How effective were all these different optimization efforts we just described? The most straightforward benchmark is one in which raw runtime (and hopefully speedup) is measured. In this case, we measured the time to complete the processing of a single DESI frame on a single Edison, Cori Haswell, and Cori KNL node. In Figure 7 we show how each optimization affected the single frame runtime. The optimizations are plotted chronologically against the overall runtime of the frame on each architecture.

Figure 7 shows that the first few changes we made had the largest overall impact: the later optimizations exhibited some diminishing returns. Over the course of this work the runtime for a single frame was decreased from 4000 to 525 seconds for KNL, from 862 to 130 seconds for Haswell, and from 1146 to 116 seconds for Ivy Bridge (the processor architecture on NERSC's now retired Edison system). The overall increases in raw speed varied between 7-10x for each architecture. One major goal of the NESAP program was to reduce the DESI runtime on KNL to below the original Edison Ivy Bridge benchmark, which is indicated by the red dotted line. Once we implemented our `legval` cache fix, we achieved this goal.

A more informative benchmark for DESI is specific processing throughput, stated in frames processed per node-hour. Measuring this quantity makes it clear how much of DESI's computing allocation is needed to complete a given amount of processing. Higher specific throughput indicates more effective use of computing resources. We measure this benchmark using a full exposure (30 frames), instead of a single frame. We also measure on either 19 or 9 nodes for Haswell and KNL, respectively, due to the limitations we described earlier (in the Parallelize over Subbundles Instead of Bundles subsection). Though a single exposure is still a relatively small test because DESI expects to collect 30 or more exposures per night (approximately 1000 frames), it much more

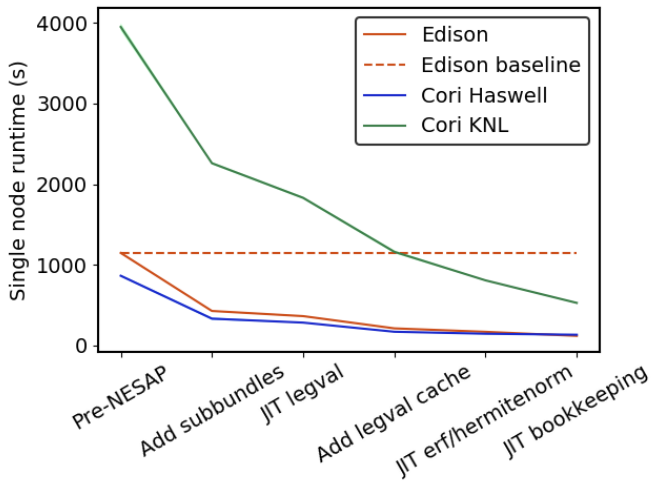


Fig. 7: *The single-node speedup achieved on Intel Ivy Bridge, Haswell, and KNL architectures throughout the course of this study.*

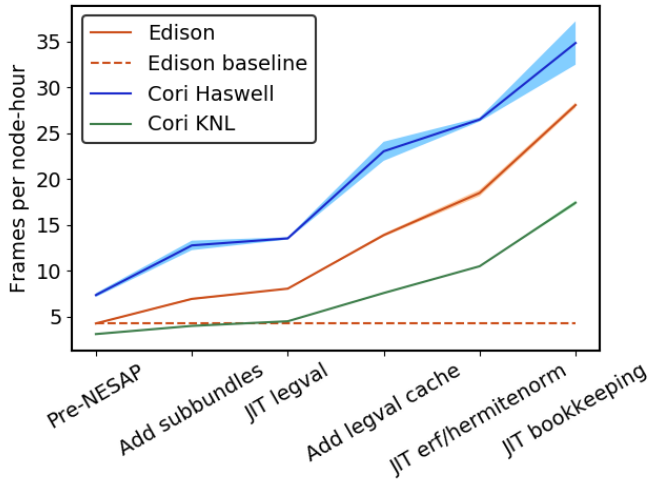


Fig. 8: *This figure shows the improvement over the course of this study in the DESI spectral extraction specific throughput.*

closely approaches the real DESI workload than the single frame benchmark. One feature encoded in this benchmark which is not captured in the speed benchmark is the increasingly important role that MPI overhead begins to play in multi-node jobs, which is a real factor with which DESI will have to contend during its large processing runs. The frames per node-hour results are plotted in Figure 8. While the increases in specific throughput we have obtained are more modest than the raw speedup, these values are a more accurate representation of the actual improvements in DESI's processing capability. For this reason we emphasize that we were able to achieve a 5-7x specific throughput increase instead of the (more exciting but less meaningful) 7-10x in raw processing speed.

It is worth mentioning that using Numba allowed us to make notable improvements specifically on KNL, which was of course the main goal of this study. For `legval` in particular, shown in Figure 6, we found that JIT compiling this function provided 15x speedup on KNL vs only 5x speedup on Haswell. This additional speedup on KNL was because Numba was able to target the KNL AVX-512 vector units. We therefore strongly recommend

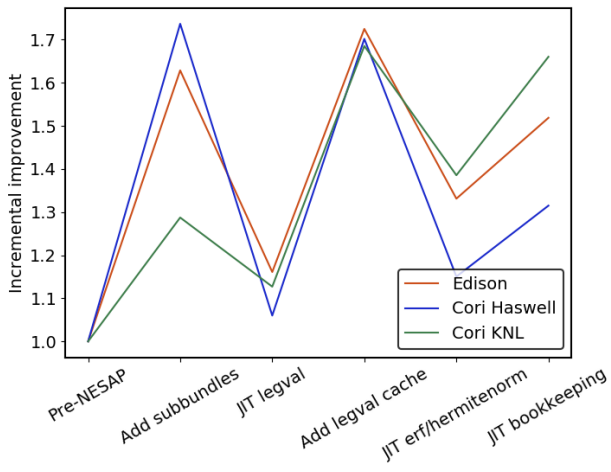


Fig. 9: *Types of optimization efforts performed in this study and their resulting incremental specific throughput improvements on Intel Ivy Bridge, Haswell, and Knights Landing architectures. These optimizations are listed in chronological order.*

investigating Numba to any developer trying to optimize Python code to run on a system with vectorization capabilities.

Finally, in Figure 9 we summarize the incremental specific throughput improvements we obtained throughout this study on Edison Ivy Bridge, Cori Haswell, and Cori KNL. The code optimizations are plotted in chronological order. Perhaps these results are the most generally instructive. First, they demonstrate that the restructuring-based optimizations were more valuable than the JIT-based optimizations. For example, the overall speedup of adding the `legval` cached values was approximately 1.7x, although this was also the most difficult of all the optimizations implemented in this study. In contrast, our relatively painless JIT compiled optimizations were not as effective in terms of speedup, averaging between a factor of 1.1-1.5x improvement. The takeaway from these results might be that if a developer has enough time, the larger, more complex restructuring optimizations may be extremely worthwhile. The flip side is that if the developer has limited time, small fixes like JIT compiling can still provide reasonable gains without a major time investment.

Alternatives to MPI?

A few problems with the current MPI implementation of the DESI spectral extraction code prompted us to take a step back and consider if newer frameworks like Dask [noaf] would be a better solution for parallelization within DESI. The reason we considered Dask, and not Apache Spark or similar frameworks, was 1) because converting to Dask would require a less extreme refactor and 2) the Dask adpatations would not preclude smaller-scale users from running DESI processing routines on their laptops, which would have been the case with Spark.

The first problem we hoped to address was the relative inflexibility of the division of work between bundles[1] . The second was the issue of resiliency: if a node goes down, it will take the entire MPI job with it[2] . An additional feature we liked about Dask is the ability to monitor Dask jobs in real time with their Bokeh status page. We thought Dask seemed promising enough that it

was worth taking a careful look at what it would mean to replace the DESI MPI with Dask.

Dask is a task-based parallelization system for Python. It is comprised of a scheduler and some number of workers which communicate with each other via a client. Dask is more flexible than traditional MPI because it can start workers and collect their results via a concurrent futures API. It should be noted that this is also possible in MPI with dynamic process management, but since Cray does not yet support dynamic process management under the Slurm workload manager, we haven't been able to try it at NERSC.

During this process, we discovered that it is non-trivial to convert a code already written in MPI to Dask, and it would likely be difficult to convert from Dask to MPI as well. (It would likely be easier to convert from dynamic process management MPI to Dask, but the DESI spectral extraction code is not written with this API.)

One major difference between MPI and Dask is the point at which the decision of how to divide the problem occurs. In MPI since all ranks are generally passing over the code, dividing the data and performing some operation on it in parallel can be done on the fly. In Dask, however, the scheduler needs to know in advance which work to assign to workers. This means that the work must already be divided in sensible way. Collecting the information required for Dask-style parallelism in advance would have required a substantial restructuring on the order of what was performed for `legval`, if not more ambitious. At this point we decided that if the DESI code had been written from the start with Dask-type parallelism in mind using Dask would have been a good choice, but converting existing MPI code into Dask was unfortunately not a reasonable solution for us.

Does it Make Sense to Run DESI Code on GPUs?

Because HPC systems are becoming increasingly heterogeneous, it is important to consider how the DESI code will run on future architectures. The next NERSC system Perlmutter [noak] will include a CPU and GPU partition that will provide a large fraction of the system's overall FLOPS, so it is pertinent to examine if and how the DESI code could take advantage of these accelerated nodes.

Since GPUs are fundamentally different from CPUs, it may be necessary to rethink much of the way in which the DESI spectral extraction is performed. At the moment, each CCD frame is divided into 7200 overlapping subregions such that each matrix to solve is typically 400x400 elements. Though this division of a larger frame into smaller pieces makes sense for CPU architectures, it may not be optimal for GPU architectures. In fact for GPUs often the opposite is true: the programmer should give the GPU as much work as possible to keep it occupied; thus it may be beneficial to operate on a smaller number of larger matrices. Additionally, it may be necessary to change the code so that the matrices are both constructed and solved on the GPU to bypass inefficient subregion bookkeeping, which is currently interleaved between constructing and solving the matrices, and avoid expensive data transfer. This means that helping the DESI extraction code run efficiently on GPUs could require a major restructuring to better adapt the problem for the capabilities of the hardware.

1. Although this is currently being addressed in the subbundle division restructure.

2. This is not an issue in Dask, in which dead workers can be seamlessly revived while the calculation continues.

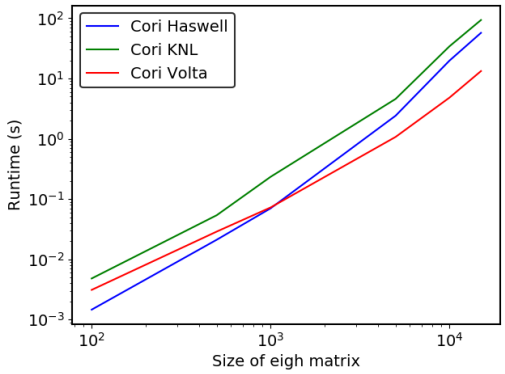


Fig. 10: *Data from performing an eigh matrix decomposition of various sizes on Edison Ivy Bridge, Cori Haswell, Cori KNL, and Cori Volta. We used CuPy to perform eigh on the Volta GPU.*

Preliminary testing is underway to give some indication of what we might expect from a major overhaul. From profiling information we expect that the `scipy.linalg.eigh` function will constitute a larger part of the workload as matrix sizes increase. We have measured the runtime of `scipy.lialg.eigh` and `cupy.linalg.eigh` [noac] as an initial test case on Cori Haswell, KNL, and the new Cori Volta GPUs. (We could not make these measurements on Edison Ivy Bridge because it has now been decommissioned.) Figure 10 shows the `eigh` runtime for various sizes of positive definite input matrices. These data show that for larger matrix sizes (above approximately 1000) the Volta begins to outperform the CPUs. However, these data do not include any possible gains from a divide-and-conquer approach (which has proven very successful for DESI). Investigating this strategy is near-term future work.

This `eigh` study is just the first of many planned GPU experiments. DESI has additional matrix preparation steps, bookkeeping, and special function evaluations (like `legval`) which also constitute a large part of their total workload. At this time it is unclear which of these might perform well on the GPU and make the relatively expensive host to device data transfer worthwhile. We will perform many experiments to evaluate how well each of these are suited to the GPU (or perhaps not suited to the GPU) as future work.

We should note that one of the major conclusions of this case study has been that large restructuring efforts have been worthwhile for DESI. If indeed we choose to embark upon another major restructure for GPUs, what is the best approach? As we have detailed above, we have had reasonably good success with Numba, which also supports GPU offloading. Other options are CuPy [noab], which aims to be a drop-in replacement for NumPy, pyCUDA [noam], and pyOpenCL [noan]. How best to support GPU offloading without having to fill the DESI code with distinct CPU and GPU blocks, and additionally to avoid being tied to a particular vendor, is still an open question for us.

Conclusions and Future Work

Over the course of this work, we have achieved our goal of speeding up the throughput of the DESI spectral extraction code on NERSC Cori Haswell and KNL processors by a factor of 5-7x without rewriting their Python code in another language.

DESI will process its data at NERSC both in semi-realtime and additionally, it will reprocess all of its data each year (at least) with the latest pipeline version. At the start of this work, the final data processing would have taken 33 million CPU hours. The work presented in this study has reduced that to 6.5 million hours, making much more efficient use of the resources available at NERSC, thus benefitting both the DESI project and also the many other users who share the NERSC systems. Additionally, this algorithm speedup lets DESI process a night's data in a matter of hours instead of days, enabling the ability to use one night of data as feedback to the survey operations the following night. This results in more efficient survey operations, reducing the time to completion.

Our strategy was as follows: we employed profiling tools, starting with the most simple tools (cProfile + gprof2dot) and progressing as necessary to more complex tools (line_profiler and Tau), to get an idea of which kernels are most expensive and what types of structural changes could help improve runtime and flexibility. We used Numba to JIT compile several expensive functions. This was a relatively quick way to obtain some speedup without changing many lines of code. We also made larger structural changes to avoid calling expensive functions and also to increase the flexibility and efficiency of the parallelism. In general these larger structural changes were more complex to implement, as well as more time consuming, but also resulted in the biggest payoff in terms of speedup.

We considered changing the parallelism strategy from MPI to Dask, but ultimately found that changing an existing code is non-trivial due to the fundamentally different strategies of dividing the workload, and decided to continue using MPI. Work is in progress to address two remaining issues: load-balancing and inflexible job size. Finally, we are now investigating how the DESI code could run effectively on GPUs by since the next NERSC system Perlmutter will include a large CPU and GPU partition. Exploratory studies for how the DESI code can be optimized are being performed using `scipy.linalg.eigh` and `cupy.linlg.eigh` as a test case now and will continue as future work.

Acknowledgments

The authors thank their partners at Intel, the Intel Python Team, Intel tools developers, performance engineers, and their management. The authors also would like to thank the Tau Performance System team at the University of Oregon for their help in building Tau for our application. This work used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Additionally, this research is supported by the Director, Office of Science, Office of High Energy Physics of the U.S. Department of Energy under Contract No. DE–AC02–05CH1123, and by the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility under the same contract; additional support for DESI is provided by the U.S. National Science Foundation, Division of Astronomical Sciences under Contract No. AST-0950945 to the National Optical Astronomy Observatory; the Science and Technologies Facilities Council of the United Kingdom; the Gordon and Betty Moore Foundation; the Heising-Simons Foundation; the National Council of Science and Technology of Mexico, and by the DESI Member Institutions. The authors are honored to be permitted to conduct astronomical research on Iolkam Du'ag (Kitt Peak), a mountain with particular significance to the Tohono O'odham Nation.

References

[adm] admin. Python* Code Analysis. URL: https://software.intel.com/en-us/vtune-amplifier-help-python-code-analysis.

[BS10] Adam S Bolton and David J Schlegel. Spectro-perfectionism: an algorithmic framework for photon noise-limited extraction of optical fiber spectroscopy. *Publications of the Astronomical Society of the Pacific*, 122(888):248, 2010. doi:10.1086/651008.

[Fon19] José Fonseca. Converts profiling output to a dot graph. Contribute to jrfonseca/gprof2dot development by creating an account on GitHub, April 2019. original-date: 2015-03-13T07:19:30Z. URL: https://github.com/jrfonseca/gprof2dot.

[Ker19] Robert Kern. Line-by-line profiling for Python. Contribute to rkern/line_profiler development by creating an account on GitHub, April 2019. original-date: 2014-08-31T11:45:05Z. URL: https://github.com/rkern/line_profiler.

[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.

[MWW13] Michael J Mortonson, David H Weinberg, and Martin White. Dark energy: a short review. *arXiv preprint arXiv:1401.0046*, 2013.

[noaa] 26.3. The Python Profilers — Python v3.2.6 documentation. URL: https://docs.python.org/3.2/library/profile.html.

[noab] CuPy. URL: https://cupy.chainer.org/.

[noac] cupy.linalg.eigh — CuPy 5.4.0 documentation. URL: https://docs-cupy.chainer.org/en/stable/reference/generated/cupy.linalg.eigh.html.

[noad] Cython: C-Extensions for Python. URL: https://cython.org/.

[noae] Dark Energy Spectroscopic Instrument (DESI). URL: https://www.desi.lbl.gov/.

[noaf] Dask: Scalable analytics in Python. URL: https://dask.org/.

[noag] National Energy Research Scientific Computing Center. URL: https://www.nersc.gov/.

[noah] NESAP. URL: https://www.nersc.gov/users/application-performance/nesap/.

[noai] Numba: A High Performance Python Compiler. URL: http://numba.pydata.org/.

[noaj] numpy.polynomial.legendre.legval — NumPy v1.16 Manual. URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.polynomial.legendre.legval.html.

[noak] Perlmutter. URL: https://www.nersc.gov/systems/perlmutter/.

[noal] Publications Resulting from the Use of NERSC Resources. URL: https://www.nersc.gov/news-publications/publications-reports/nersc-user-publications/.

[noam] PyCUDA. URL: https://mathema.tician.de/software/pycuda/.

[noan] PyOpenCL. URL: https://mathema.tician.de/software/pyopencl/.

[noao] scipy.special.erf — SciPy v0.14.0 Reference Guide. URL: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.special.erf.html.

[noap] scipy.special.hermitenorm — SciPy v1.2.1 Reference Guide. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.hermitenorm.html.

[noaq] SnakeViz. URL: https://jiffyclub.github.io/snakeviz/.

[noar] TAU - Tuning and Analysis Utilities -. URL: https://www.cs.uoregon.edu/research/tau/home.php.

[PR03] P James E Peebles and Bharat Ratra. The cosmological constant and dark energy. *Reviews of modern physics*, 75(2):559, 2003. doi:10.1086/185100.

[PTV+92] William H Press, Saul A Teukolsky, William T Vetterling, Brian P Flannery, and M Metcalf. Numerical recipes in Fortran 90. *The art of scientific computing,(Cambridge, 1996)*, 1992. doi:10.1109/MCC.1998.736436.

[RTD+17] Zahra Ronaghi, Rollin Thomas, Jack Deslippe, Stephen Bailey, Doga Gursoy, Theodore Kisner, Reijo Keskitalo, and Julian Borrill. Python in the NERSC Exascale Science Applications Program for Data. In *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing - PyHPC'17*, pages 1–10, Denver, CO, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3149869.3149873, doi:10.1145/3149869.3149873.

A Real-Time 3D Audio Simulator for Cognitive Hearing Science

Mark Wickert‡*

http://www.youtube.com/watch?v=dhRUe-gz690

Abstract—This paper describes the development of a 3D audio simulator for use in cognitive hearing science studies and also for general 3D audio experimentation. The framework that the simulator is built upon is `pyaudio_helper`, which is a module of the package `scikit-dsp-comm`. The simulator runs in a Jupyter notebook and makes use of Jupyter widgets for interactive control of audio source positioning in 3D space. 3D audio has application in virtual reality and in hearing assistive devices (HAD) research and development. At its core the simulator uses digital filters to represent the sound pressure wave propagation path from the sound source to each ear canal of a human subject. Digital filters of 200 coefficients each for left and right ears are stored in a look-up table as a function of azimuth and elevation angles of the impinging sound's source.

Index Terms—Head-related impulse response (HRIR), Head-related transfer function (HRTF), binaural hearing, virtual reality, audiology, hearing assistive devices (HAD),

Introduction

In cognitive hearing science binaural hearing models how sound pressure waves arrive at either ear drum, at the end of the ear canal, or in the case of typical measurements, at the entry to the ear canal, both as a function of the arrival angle in 3D (azimuth and elevation) and radial distance. A tutorial on 3-D audio can be found at [HCI]. This leads to the need for the head related impulse response (HRIR) (time-domain) or head-related transfer function (HRTF) (frequency domain) for a particular human subject. Traditionally human subjects are placed in an anechoic chamber with a sound source placed at e.g. one meter from the head and then moved relative the subject's head over a range of azimuth and elevation angles, with the HRIR measured at each angle. The 3D simulator described here uses a database of HRIR's from the University of California, Davis, originally in the Center for Image Processing and Integrated Computing (CIPIC), [CIPICHRTF], to describe a given subject. In the `pyaudio_helper` application the HRIR at a given angle is represented by two (left and right ear) 200 coefficient digital filters that the sound source audio is passed through. Here the data base for each subject holds 25 azimuth and 50 elevation angles to approximate continuous sound source 3D locations.

* *Corresponding author: mwickert@uccs.edu*
‡ *University of Colorado Colorado Springs*

Obtaining individual HRTFs is a challenge in itself and the subject of much research.

In a related research project *deep learning* is being investigated as a means to fit a human subject to the CIPIC HRTF database of subjects, based on 27 upper torso anthropometrics (measurements) of the subject. As a simple solution, we can also consider using a simple spherical head model, and its corresponding HRTF, which makes use of spherical harmonics to solve for the sound pressure magnitude and phase at any location on the sphere surface. A frequency sweep of magnitude and phase is then inverse Fourier transformed to obtain the HRIR. The ultimate intent of the simulator is to serve as a clinical tool for blind sound source localization experiments. Human subjects will be exposed to several different HRIR models, where at least one model is a *personalized fit* based on deep learning using anthropometrics and/or a finite element wave equation solution using a 3D rendering of the subject's shoulders and head. 3D rendering of a subject can be obtained using *photogrammetry*, which estimates three-dimensional coordinates of points on an object from a collection of photographic images taken from different positions.

3D Geometry

To produce a synthesized 3D audio sound field, we start with a geometry where the center of the coordinate frame is the intersection between the subject's *mid-sagittal* or vertical *median plane* and the line connecting the left and right ear canals. This is referred to as being *head-centered*. The coordinate systems used in this paper are shown in Figure 1. The primary head-centered system has cartesian coordinates labeled (x,y,z) and associated cylindrical coordinates $(r_{xy},\phi_{\text{az}},h_y)$ (black labels in Figure 1). The cylindrical coordinates will be used in Jupyter notebook apps presented later as the interface for GUI controls to conveniently position the audio source about a subject's head. A secondary head-centered system, used by CIPIC, has cartesian coordinates labeled (x_1,x_2,x_3) and associated spherical coordinates (r,ϕ,θ) (purple labels in Figure 1). The first coordinate system is motivated by [Fitzpatrick], and its usage is explained in detail in the section FIR Filter Coefficient Set Selection. The second system is referred to by CIPIC as the *interaural-polar coordinate system* (IPCS), which is used to index into the HRIR filter pairs which produce the right and left audio outputs.

The 3D audio rendering provided by the simulator developed in this paper relies on the 1250 HRIR measurements taken using the geometrical configuration shown in Figure 2. A total of 45 subjects are contained in the CIPIC HRIR database, both human

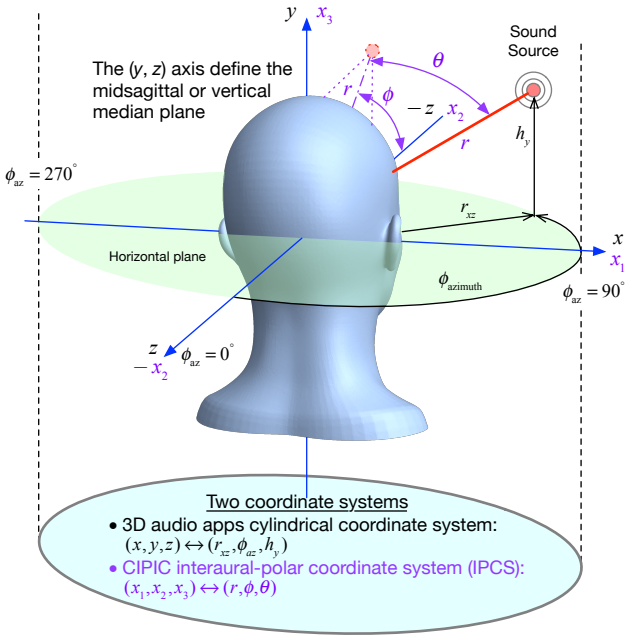


Fig. 1: *The primary head-centered coordinate system, (x,y,z), used in the 3D audio simulator, along with the secondary system, (x_1,x_2,x_3) used by CIPIC via IPCS and spherical coordinates (r,ϕ,θ).*

and the mannequin KEMAR (Knowles Electronics Manikin for Auditory Research) [CIPICHRTF]. For subject 165 in particular, the left-right channel HRIR is shown in Figure 3, for a particular cylindrical coordinate system triple (r_{xz},h_y,ϕ_{az}). Figure 3 in particular illustrates two binaural cues, *interaural level difference* ILD and *interaural time difference* ITD, that are used for accurate localization of a sound source. With $\phi_{az} = 130^\circ$ we see as expected, the impulse response for the right ear arriving ahead of the left ear response, and with greater amplitude.

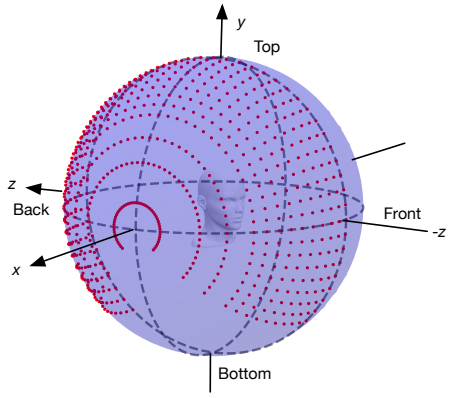


Fig. 2: *The CIPIC audio source locations, effectively on a 1 m radius sphere, used to obtain 1250 HRIR measurements for each of 45 subjects (only the right hemisphere locations shown).*

Real-Time Signal Processing

In this section we briefly describe the role real-time digital signal processing (DSP) plays in implementing the 3D audio simulator. A top level block diagram of the 3D audio simulator is shown in Figure 4. For an audio source positioned at (x,y,z) relative to the head center, the appropriate HRIR right and left channel digital

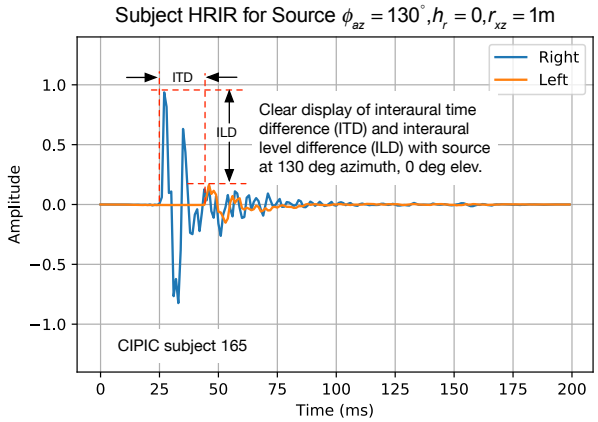


Fig. 3: *Example right/left HRIR plots for a particular arrival angle pulled from CIPIC for subject 165.*

filter coefficients are utilized along with gain scaling to account for radial distance relative to 1 m and a parallax correction factor. Gain scaling and parallax correction, are taken from [Fitzpatrick], and are explained in more detail in the following section of this paper.

To implement the filtering action we use the `pyaudio_helper` framework [Wickert] of Figure 5, which interfaces to the audio subsystem of a personal computer. The framework supports real-time signal processing, in particular filtering using core signal processing functions of `scipy.signal` [ScipySignal]. The 200 coefficients of the right and left HRIR are equivalent to the coefficients in a finite impulse response (FIR) digital filter which produce a discrete-time output signal or sequence $y_R[n]/y_L[n]$ from a single audio source signal $x[n]$. All of the signals are processed with at a sampling rate of $f_s = 44.1$ kHz, as this is rate used in forming the CIPIC database. In mathematical terms we have the output signals that drive

$$y_R[n] = G_R \sum_{m=0}^{M} b_R x[n-m] \quad (1)$$

$$y_L[n] = G_L \sum_{m=0}^{M} b_L x[n-m] \quad (2)$$

where G_R and G_L are right/left gain scaling factors that take into account the source distance relative to the 1 m distance used in the CIPIC database and b_R and b_L are the right/left HRIR coefficient sets appropriate for the source location.

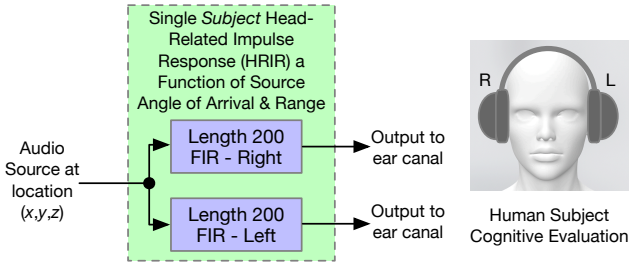


Fig. 4: *Real-time DSP filtering with coefficients determined by the audio source (x,y,z) location.*

To produce real-time filtering with `pyaudio_helper` requires [Wickert] (i) create an instance of the `DSP_io_stream`

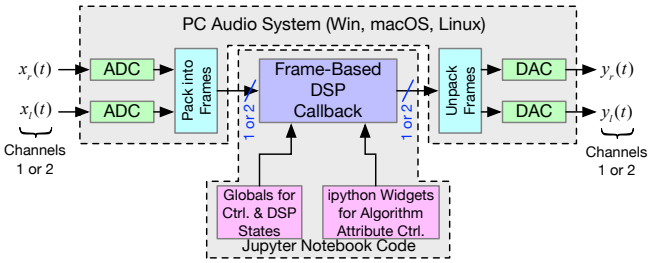


Fig. 5: *The pyaudio_helper framework for real-time DSP in the Jupyter notebook.*

class by assigning valid PC audio input and output device ports to it, (ii) define a `callback` function to process the input signal sample frames into right/left output sample frames according to (1), and (iii) call the method `interactive_stream()` to start streaming. All of the code for the 3D simulator is developed in a Jupyter notebook for prototyping ease. Since [Wickert] details steps (i)-(iii), in the code snippet below we focus on the key filtering expressions in the callback and describe the playback of a geometrically positioned *noise* source via headphones:

```
def callback(in_data, frame_length, time_info,
             status):
    global ...
    ...
    #********************************************
    # DSP operations here:
    # Apply Kemar HRIR left and right channel
    # filters at the sound source location in
    # cylindrical coordinates mapped to cartesian
    # coordinates from GUI sliders
    # The input to both filters comes by first
    # combining x_left & x_right channels or here
    # input white noise
    x_mono = Gain.value*5000*randn(frame_length)
    subj.cart2ipcs(r_xz_plane.value*sin(pi/180* \
                   azimuth.value), #x
                   y_axis.value,   #y
                   r_xz_plane.value* \
                   cos(pi/180* \
                   azimuth.value)) #z
    # Filter a frame of samples and save initial
    # conditions for the next frame
    y_left, zi_left = signal.lfilter(subj.coeffL,
                                     1,subj.tL*x_mono,
                                     zi=zi_left)
    y_right, zi_right = signal.lfilter(subj.coeffR,
                                     1,subj.tR*x_mono,
                                     zi=zi_right)
    #********************************************
    ...
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue

# Create a ss_mapping2CIPIChrir object
# SUBJECT 20, 21 (KEMAR SM ears),
# & 165 (KEMAR LG ears)
# subject_200, 201 is 8.75 cm, 10 cm sphere
subj = ss_mapping2CIPIChrir('subject_165')
# Initialize L/R filter initial conditions
zi_left = signal.lfiltic(subj.coeffL,1,[0])
zi_right = signal.lfiltic(subj.coeffR,1,[0])
# Create a IO stream object and start streaming
DSP_IO = pah.DSP_io_stream(callback,0,1,
                           frame_length=1024,
                           fs=44100,Tcapture=0)
DSP_IO.interactive_stream(0,2)
# Show Jupyter widgets
widgets.HBox([Gain,r_xz_plane,azimuth,y_axis])
```

FIR Filter Coefficient Set Selection

To finally render 3D audio requires selection of the appropriate right/left filter coefficient set, and if needed range correction. For the special case of the source on the 1 m CIPIC reference sphere, we simply pick the coefficient set that lies closest to the desired IPCS angle pair (ϕ, θ).

For the more typical case of the source range, $r = \sqrt{x^2+y^2+z^2} \neq 1$, more processing is required. The approach taken here follows the methodology of [Fitzpatrick] by using the primary cartesian coordinates of Figure 1 to additionally perform *parallax* correction and source range amplitude correction. At distance r from a point source the sound wave energy diverges by $1/r^2$, so in terms of wave amplitude we include a scale factor of $1/r$. Here the inverse distance correction also takes into account the fact that the entry to the ear canal is offset from the head center by the mean head radius R. The second correction factor is *parallax*, which is graphically depicted in Figure 6 for the special case of a source in the horizontal plane and directly in front of the head. Both corrections are addressed in detail in [Fitzpatrick]. For a source not on the unit sphere, sound parallax requires an adjustment in the HRIR coefficients, unique to the right and left ears. If we extend rays from the right and left ears that pass through the sound source location and then touch the unit sphere, the required azimuth values will be shifted to locations on either side of the true source azimuth. The corresponding HRIR values where these rays contact the unit sphere, respectively, perform the needed parallax correction. The actual database entries utilized are those that are closest to the intersection points.

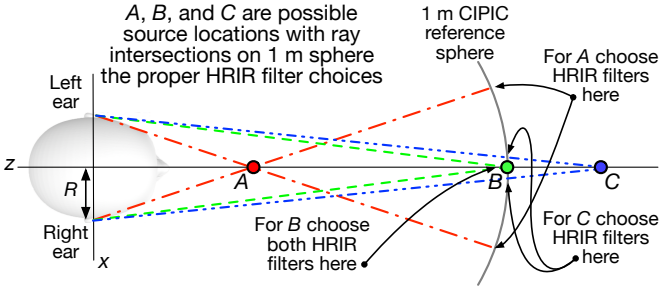


Fig. 6: *Parallax correction geometry for three possible source locations in the horizontal plane: $A < 1$ m, $B = 1$ m, and $C > 1$ m, directly in front of the head.*

The class `ss_mapping2CIPIChrif()` takes the source location, (x,y,z), and using the single method `cart2ipcs(self,x,y,z)`, produces the parallax corrected right and left HRIR filter coefficients and range amplitude scaling factors. The code is listed below:

```
class ss_mapping2CIPIChrir(object):
    """
    A class for sound source mapping to the CIPIC
    HRIR database

    CIPIC uses the interaural polar coordinate
    system (IPCS). The reference sphere for the
    head-related transfer function (HRTF)
    measurements/head-related impulse response
```

```
    (HRIR) measurements has a 1m radius.

    Mark Wickert June 2018

def __init__(self,sub_foldername,
             head_radius_cm = 8.75):
    """
    Object instantiation

    The default head radius is 8.75 cm
    """
    # Store the head radius in meters
    self.head_radius = head_radius_cm/100

    # Store the HRIR 200 tap FIR filter coef sets
    self.subject = sub_foldername
    hrir_LR = io.loadmat( self.subject + \
                          '/hrir_final.mat')
    self.hrirL = hrir_LR['hrir_l']
    self.hrirR = hrir_LR['hrir_r']

    # Create LUTs for the azimuth and elevation
    # values. This will make it easy to quantize
    # a given source location to one of the
    # available HRIRs in the database.
    self.Az_LUT = np.hstack(([-80,-65,-55],
                  np.arange(-45,45+5,5.0),
                            [55,65,80]))
    self.El_LUT = -45 + 5.625*np.arange(0,50)

    # Initialize parameters
    self.tR = 1 # place source on unit sphere
    self.tL = 1 # directly in front of listener
    self.elRL = 0
    self.azR = 0
    self.azL = 0
    self.AzR_idx = 0
    self.AzL_idx = 0
    self.ElRL_idx = 0

    # Store corresponding right and left ear FIR
    # filter coefficients
    self.coeffR = self.hrirR[0,0,:]
    self.coeffL = self.hrirL[0,0,:]


def cart2ipcs(self,x,y,z):
    """
    Map cartesian source coordinates (x,y,z) to
    the CIPIC interaural polar coordinate system
    (IPCS) for easy access to CIPIC HRIR. Parallax
    error is also dealt with so two azimuth values
    are found. To fit IPCS the cartesian
    coordinates are defined as follows:

    (0,0,0) <--> center of head.
    (1,0,0) <--> unit vector pointing outward from
                 the right on a line passing from
                 left to right through the left
                 and right ear (pinna) ear canals
    (0,1,0) <--> unit vector pointing out through
                 the top of the head.
    (0,0,1) <--> unit vector straight out through
                 the back of the head, such that
                 a right-handed coordinate system is
                 formed.

    Mark Wickert June 2018, updated June 2019
    """
    # First solve for the parameter t, which is used
    # to describe parametrically the location of the
    # source at (x,y,z) on a line connecting the
    # right or left ear canal entry point to the
    # unit sphere.

    # The right ear (pinna) solution
    aR = (x-self.head_radius)** + y**2 + z**2
    bR = 2*self.head_radius*(x-self.head_radius)
    cRL = self.head_radius**2 - 1
    # The left ear (pinna) solution
    aL = (x+self.head_radius)**2 + y**2 + z**2
    bL = -2*self.head_radius*(x+self.head_radius)

    # Find the t values which are also the gain
    # values to be applied to the filter.
    self.tR = max((-bR+np.sqrt(bR**2-4*aR*cRL)) \
              /(2*aR),
             (-bR-np.sqrt(bR**2-4*aR*cRL))/(2*aR))
    self.tL = max((-bL+np.sqrt(bL**2-4*aL*cRL)) \
              /(2*aL),
             (-bL-np.sqrt(bL**2-4*aL*cRL))/(2*aL))
    # Find the IPCS elevation angle and mod it
    elRL = 180/np.pi*np.arctan2(y1,-z1)
    if elRL < -90:
          elRL += 360
    self.elRL = elRL
    self.azR = 180/np.pi* \
               np.arcsin(np.clip(self.head_radius\
                + self.tR*(x1-self.head_radius),
                -1,1))
    self.azL = 180/np.pi* \
               np.arcsin(clip(-self.head_radius\
                + self.tL*(x1+self.head_radius),
                -1,1))
    # Find closest database entry in Az & El
    self.AzR_idx = np.argmin((self.Az_LUT \
                             - self.azR)**2)
    self.AzL_idx = np.argmin((self.Az_LUT \
                             - self.azL)**2)
    self.ElRL_idx = np.argmin((self.El_LUT \
                             - self.elRL)**2)
    self.coeffR = self.hrirR[self.AzR_idx,
                             self.ElRL_idx,:]
    self.coeffL = self.hrirL[self.AzL_idx,
                             self.ElRL_idx,:]
```

In the `__init__` method all the right left filter coefficients for the chosen subject database entry are copied into class attributes and look-up tables (LUTs) are populated in terms of IPCS angles to ease selecting the needed right/left filters. Note in particular the scale factors `self.tR` and `self.tL` are the inverse distance wave amplitude correction factors representing G_R and G_L in (1) and (2), respectively.

3D Audio Simulator Notebook Apps

For human subject testing and general audio virtual reality experiments, two applications (apps) that run in the Jupyter notebook were created. The first allows the user to *statically* locate an audio source in space, while the second creates a *time-varying motion* audio source. For human subject tests the static source is of primary interest. Both apps have a GUI slider interface that use the cylindrical coordinates described in Figure 1 to control the position the source.

Static Sound Source

The first and foremost purpose of the 3D audio simulator is to be able to statically position an audio source and then ask a human subject where the source is located (localization). This is a cognitive experiment, and can serve many purposes. One purpose in the present research is to to see how well the HRIR utilized in the simulator matches the subject's true HRIR. As mentioned in the introduction, an ongoing study is to estimate an *individualized HRIR* using deep machine learning/deep learning. The Jupyter Widgets slider interface for this app is shown in Figure 7

Dynamic Sound Source Along a Trajectory

From a virtual reality perspective, we were also interested in giving a subject a moving sound source experience via headphones.

```
# Create a ss_mapping2CIPIChrir object
# SUBJECT 20, 21 (KEMAR sm), & 165 (KEMAR LG) available now
subject = ss_mapping2CIPIChrir('subject_165')
# Initialize L/R filter initial conditions
zi_left = signal.lfiltic(subject.coeffL,1,[0])
zi_right = signal.lfiltic(subject.coeffR,1,[0])
# Create a IO stream object and start streaming
DSP_IO = pah.DSP_io_stream(callback,0,1,frame_length=1024,
                           fs=44100,Tcapture=0)
DSP_IO.interactive_stream(0,2)
widgets.HBox([Gain,r_xz_plane,azimuth,y_axis])
```

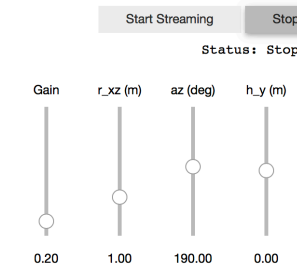


Fig. 7: *Jupyter notebook for static positioning of the audio test source.*

```
# Create a ss_mapping2CIPIChrir object
# SUBJECT 20, 21 (KEMAR sm), & 165 (KEMAR LG) available now
subject = ss_mapping2CIPIChrir('subject_165')
# Initialize L/R filter initial conditions
zi_left = signal.lfiltic(subject.coeffL,1,[0])
zi_right = signal.lfiltic(subject.coeffR,1,[0])
# Create a IO stream object and start streaming
DSP_IO = pah.DSP_io_stream(callbackTraj,0,1,frame_length=1024,
                           fs=44100,Tcapture=0)
DSP_IO.interactive_stream(0,2)
widgets.HBox([Gain_T,Period_T,r_xz_T,theta_roll_T,theta_pitch_T,h_y_T])
```

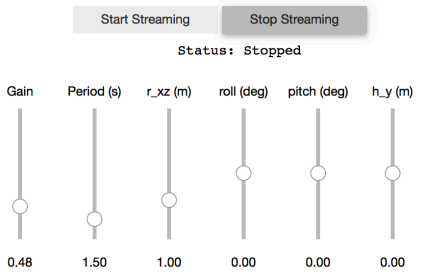


Fig. 9: *Jupyter notebook for setting the parameters of a sound source moving along a trajectory with prescribed motion characteristics.*

In this case we consider an *orbit like* sound source trajectory. The trajectory as shown in Figure 8, is a circular orbit with parameters of roll, pitch, and hight, relative to the ear canal centerline. The Jupyter Widgets slider interface for this app is shown in Figure 9.

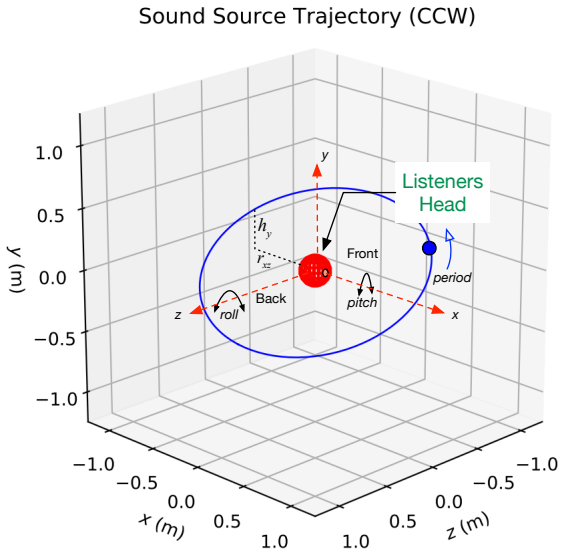


Fig. 8: *The sound source trajectory utilized in the dynamic sound source app.*

Spherical Head Model as a Simple Reference HRIR

In blind testing of human subjects it is also of interest to offer other HRIR solutions, e.g., the [KEMAR] mannequin head or a simple spherical head [Duda] and [Bogelein]. In this section we consider a spherical head model with the intent of using the results of [Duda] to allow the construction of a CIPIC-like database entry, that can be used in the 3D audio simulator described earlier in this paper.

General Pressure Wave Solution

As a starting point, the acoustics text [Beranek], provides a solution for the resultant sound pressure at any point in space when a sinusoidal plane wave sound pressure source impinges upon a rigid sphere of radius R, centered at the coordinate system origin. Rotationally symmetric spherical coordinates, r and θ are appropriate here. First consider the incident plane wave $\tilde{p}_I(r,\theta)$, in the expansion

$$\tilde{p}_I(r,\theta_i) = \tilde{p}_0 \sum_{n=0}^{\infty} (-j)^n (2n+1) j_n(kr) P_n(\cos\theta_i), \tag{3}$$

where θ_i is the incidence angle between the plane wave and measurement point, $P_m(x)$ is the nth-order Legendre polynomial, $j_n(x)$ is the nth-order spherical Bessel function of the first kind, $k = 2\pi f/c$ is the wavenumber, with f frequency in Hz and $c = 344.4$ m/s the propagation velocity in air. We set the incident wave complex pressure $\tilde{p}_0 = 1\angle 0^\circ$ for convenience.

Finally, solve for the scattered wave, $\tilde{p}_s(r,\theta_i)$, by applying boundary conditions, see [Beranek] for details. The resultant wave is the sum of the incident and scattered waves as given below:

$$\begin{aligned} \tilde{p}(r,\theta_i) &= \tilde{p}_I(r,\theta_i) + \tilde{p}_s(r,\theta_i) \\ &= \sum_{n=0}^{\infty} (-j)^n (2n+1) P_n(\cos\theta_i) \\ &\quad \cdot \left[j_n(kr) - \frac{j_n'(kR)}{h_n'^{(2)}(kR)} h_n^{(2)}(kr) \right] \end{aligned} \tag{4}$$

where $j_n'(x)$ is the spherical Bessel function of the first kind derivative, $h_n^{(2)}(kr)$ is the nth-order spherical Hankel function of the second kind, and $h_n'^{(2)}(kr)$ is the corresponding derivative. Figure 10 shows the pressure magnitude at 2000 Hz for $R = 8.75$ cm, for the plane wave traveling along the $+z-$axis. For plotting convenience, the axes z and $w = \sqrt{x^2+y^2}$ are cylindrical coordinates, as the sphere has axial symmetry. To be clear z and w are related to the original spherical coordinates of the math model by $r = \sqrt{w^2+z^2}$ and $\cos\theta_i = z/\sqrt{w^2+z^2}$.

The calculations required to evaluate (4), and thus create the plot of Figure 10, conveniently make use of functions in `scipy.special`. This is shown in the code listing below:

```
def pS(w, z, f, R = 0.0875, N = 50):
    """
    Scattered field from a rigid sphere

    w = radial comp in cylind coord
```

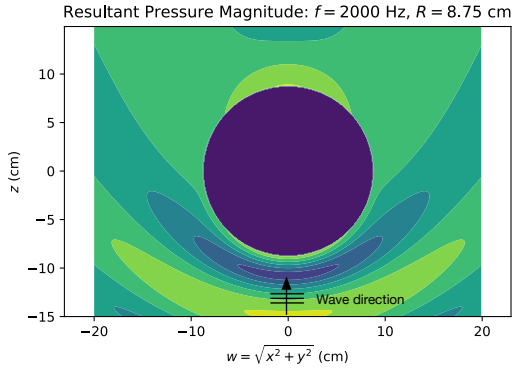


Fig. 10: *The resultant sound pressure wave magnitude in cylindrical coordinates z and w, due to scattering of a plane wave from a rigid sphere.*

```
    z = axial comp in cylind coord
    f = frequency in Hz
    R = sphere radius in m
    N = summation upper boundary

    p_polar = pressure in Pa for p0 = 1 Pa

    Mark Wickert November 2018
    """
    p_0 = 1
    k = 2*pi/(344.4/f)
    p_polar = zeros((len(z),len(w)),
                    dtype=complex128)
    for n,wn in enumerate(w):
        for m,zm in enumerate(z):
            r = sqrt(zm**2 + wn**2)
            cos_theta = zm/sqrt(zm**2 + wn**2)
            for kk in range(N+1):
                if r <= R:
                    p_polar[m,n] = 0.0
                else:
                    p_polar[m,n] += p_0*(-1j)**kk *\
                    (2*kk+1) * \
                    special.spherical_jn(kk,
                    k*R,True)/spherical_hn2(kk,
                    k*R,True) * spherical_hn2(kk,
                    k*r) * \
                    special.lpmv(0,kk,cos_theta)
    return -p_polar

def spherical_hn2(n,z,derivative=False):
    """ Spherical Hankel Function 2nd Kind """
    return special.spherical_jn(n,z,deriv=False) \
           -1j * special.spherical_yn(n,z,
           derivative=False)
```

The use of $R = 8.75$ cm is motivated by the *standard head* radius discussed in [Duda]. It is interesting to note that there is a *bright spot* on the back side ($\theta_i = 180°$) due to constructive interference between the waves traveling around either side of the sphere.

HRTF on the Sphere Surface

In signal processing, the *transfer function*, $H(f) = |H(f)|e^{j\angle H(f)}$, is a ratio of two complex numbers as a function frequency in Hz. In the denominator we have the magnitude and phase (angle) of the sinusoidal signal input to a system and in the numerator we have the magnitude and phase of the corresponding output signal (measurement point on the sphere or ultimately the ear canal). For the case of the HRTF the output is the sound pressure magnitude and phase at the entrance to the right and left ear canals. In the case of the CIPIC database the location of the source is at a particular azimuth and elevation on a 1 m sphere centered over the head. The HRTF of a sphere is defined more generally as the output can be any point on the surface of the sphere. The input location is generally at some distance r from the center of the sphere.

In [Duda] the HRTF is defined as the ratio of the sound pressure on the surface of the sphere divided by the pressure at the sphere center, given that the sphere *is not* present:

$$H(\theta_i, f, r, R) = \frac{r}{kR^2} e^{jkr} \sum_{n=0}^{\infty} (2n+1) P_n(\cos\theta_i) \frac{h_n^{(2)}(kr)}{h_n'^{(2)}(kR)}, \; r > R \tag{5}$$

where θ_i is the angle of incidence between the source and measurement point, f is the operating frequency in Hz, r is the distance from the source to the center of the sphere, and once again R is the sphere radius. Recall also that the wave number k contains f.

Formally this transfer function definition should include the propagation delay time from the source location r to the sphere center, but this is a *linear phase* of the form $\exp(-j2\pi fr/c)$ that can be dealt with as a time shift once the inverse Fourier transform is used to obtain the HRIR. Later we set $r = 1$ m to match the CIPIC source location relative to the head center.

An efficient algorithm for the calculation of (5) is presented in [Duda], requiring no special functions as a result of using special function recurrence relationships. The Python implementation, shown below, also incorporates an error threshold for terminating the series approximation:

```
def HRTF_sph(theta, f, r = 1.0, R = 0.01, c = 344.4,
             threshold = 1e-6):
    """
    HRTF calculation for a rigid sphere with source
    r meters from the sphere center

    Coded from pseudo-code to Python by Mark Wickert

    Reference: Appendix A of J. Acoust. Soc. Am.,
    Vol. 104, No. 5, November 1998 R. O. Duda and
    W. L. Martens: Range dependence of the response
    of a spherical head model.
    """
    x = np.cos(theta*np.pi/180)
    mu = (2 * np.pi * f * R)/c
    rho = r/R
    zr = 1/(1j * mu * rho)
    zR = 1/(1j * mu)
    Qr2 = zr
    Qr1 = zr * (1 - zr)
    QR2 = zR
    QR1 = zR * (1 - zR)
    P2 = 1
    P1 = x
    summ = 0
    term = zr/(zR * (zR - 1))
    summ += term
    term = (3 * x * zr * (zr - 1) )/ \
           (zR * (2 * zR * (zR - 1) + 1))
    summ += term;
    oldratio = 1
    newratio = np.abs(term)/np.abs(summ)
    m = 2
    while (oldratio > threshold) or \
          (newratio > threshold):
        Qr = -(2 * m - 1) * zr * Qr1 + Qr2
        QR = -(2 * m - 1) * zR * QR1 + QR2
        P = ((2 * m - 1) * x * \
             P1 - (m - 1) * P2)/m
        term = ((2 * m + 1) * P * Qr)/((m + 1) \
               * zR * QR - QR1)
        summ += term
        m += 1
```

```
        Qr2 = Qr1
        Qr1 = Qr
        QR2 = QR1
        QR1 = QR
        P2 = P1
        P1 = P
        oldratio = newratio
        newratio = np.abs(term)/np.abs(summ)
    # conjugate to match traveling wave convention
    H = np.conj((rho * np.exp(-1j * mu) * summ)/\
                (1j * mu))
    return H
```

HRIR on the Sphere Surface

The next step is to calculate the impulse response $h(t)$ corresponding to $H(f)$ via the inverse Fourier transform of the HRTF. Since we are working with digital (discrete-time) signal processing, the inverse discrete Fourier transform (IDFT) is used here, as opposed to the Fourier integral. We take samples of the HRTF at uniformly spaced frequency samples, Δf, running from 0 to one half the CIPIC sampling rate, $f_s = 44.1$kHz. This makes $h(t) \to h(n/f_s) = h[n]$ in the Python implementation shown below:

```
def freqr2imp(H,win_att = 100):
    """
    Transform the frequency response of a real
    impulse response system back to the impulse
    response, with smoothing using a window
    function.

    Mark Wickert, May 2019
    """
    Nmax = len(H)
    if win_att == 0:
        h = np.fft.irfft(H)
    else:
        W = signal.windows.chebwin(2*Nmax,
                    win_att,sym=True)[Nmax:]
        h = np.fft.irfft(H*W)
    return h


def compute_HRIR(theta_deg, r = 1.0, R = 0.0875,
          fs = 44100, roll_factor = 20):
    """
    HRIR for rigid sphere at incidence angle
    theta_deg, distance r and radius R using
    sampingrate fs Hz

    Mark Wickert, June 2019
    """
    fs = 44100
    Nfft = 2**10
    df = fs/Nfft
    f = np.arange(df,fs/2,df)
    df = fs/Nfft
    f = np.arange(df,fs/2,df)
    HRTF = np.zeros(len(f),dtype=np.complex128)
    for k, fk in enumerate(f):
        HRTF[k] = HRTF_sph(theta_deg,fk,r=r,R = R)
    # Set DC value to 1
    HRTF = np.hstack(([1],HRTF))
    f = np.hstack(([0],f))

    HRIR = freqr2imp(HRTF,win_att=100)
    # Scale HRIR so the area is unity
    G0 = 1/(np.sum(HRIR)*1/fs)
    t = np.arange(len(HRIR))/fs*1000
    return t, np.roll(G0*HRIR,roll_factor)
```

We choose Δf to obtain at least 100 samples on $[0, f_s/2]$, so that when `np.fft.irfft()` is employed, the full real impulse response length will be 200. The function *freq2imp*() also includes frequency domain windowing, via `signal.windows.chebwin()` to provide some smoothing to the discrete-time approximation. In Figure 11 we show a collection of HRIR plots, created using `HRTF_sph()`, for the source 1 m away from the center of a 8.75 cm radius sphere.

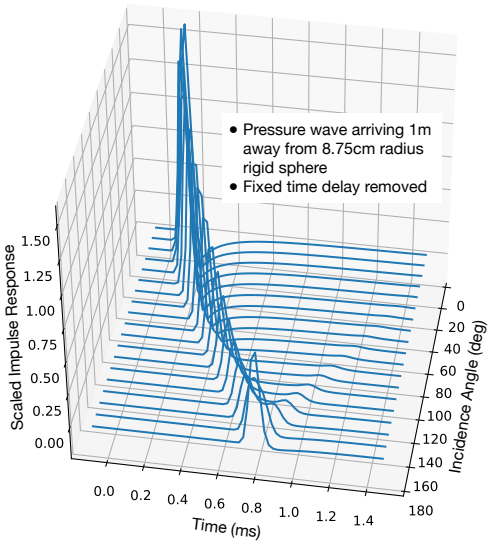


Fig. 11: *Using the spherical harmonics formulation of [Duda] to obtain the HRTF and then the HRIR as a function of sound source incidence angle from* 0° *to* 180°.

Building a CIPIC Database Entry

To finally create a CIPIC-like database entry for a spherical head, we have to relate the angle of incidence in the HRTF expression (5) to the angle of arrival of an audio source on the CIPIC 1 m sphere of Figure 2, relative to right and left ear canal entries at $\phi_{az} = \pm 80^\circ$ (a set back of $\pm 100^\circ$ from the front). The problem is depicted in Figure 12. This problem turns out to be a familiar analytic geometry problem, that of finding the angle between two 3D vectors passing through the origin, e.g.

$$\theta_{\vec{S}\vec{R}} \quad = \quad \cos^{-1}\left(\frac{\vec{S}\cdot\vec{R}}{|\vec{S}|\,|\vec{R}|}\right) = x_S \sin\phi_R + z_S \cos\phi_R \qquad (6)$$

where $\vec{R}$ is the vector to the right ear canal with angle ϕ_R, assumed to lie in the horizontal plane, and $\vec{S}$ is the vector to the source of length 1 m with primary coordinate system components (x_S, y_S, z_S) as defined in Figure 1. A similar relation holds for the left ear canal entry.

We need to fill the database using the CIPIC angle of arrival source grid using the secondary (ICPS) coordinate system. The coordinate conversion between x_S and z_s and the IPCS is $x_s = r\sin\theta_{\text{CIPIC}}$ and $z_s = -r\cos\phi_{\text{CIPIC}}\cos\theta_{\text{CIPIC}}$, so with $r = 1$ the angle of incidence formula (6) in final form is

$$\theta_{\vec{S}\vec{R}} = \sin\theta_{\text{CIPIC}}\sin\phi_R - \cos\phi_{\text{CIPIC}}\cos\theta_{\text{CIPIC}}\cos\phi_R \qquad (7)$$

and similarly for the left ear canal.

The steps for producing the HRIR filter pair over 1250 IPCS angle pairs is summarized in Figure 13.

Finally putting this all together, code was written in a Jupyter notebook to generate a CIPIC-like database entry, using `scipy.io` to write a MATLAB `mat` file, e.g., `subject_200` is a spherical head, with no ears (pinna), containing two HRIR arrays:

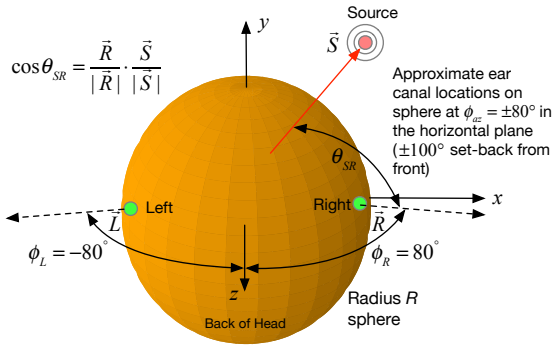


Fig. 12: *Solving for the angle between the source and a ray extending from the right and left ears, also showing a set back of the ear canal by $\pm 100°$ from the from the font of the head.*

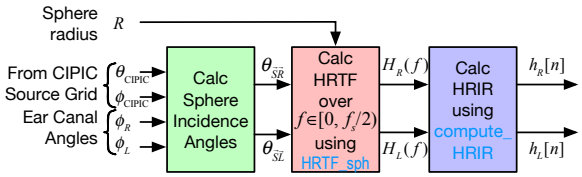


Fig. 13: *A block diagram depicting the steps involved in calculating the HRIR right and left channel impulse responses, $h_R[n]$ and $h_L[n]$, starting from CIPIC source angles, $(\theta_{CIPIC}, \phi_{CIPIC})$, ear canal set-back angles, (ϕ_R, ϕ_L), and the sphere radius R.*

```
io.whosmat('subject_200/hrir_final.mat')

[('hrir_l', (25, 50, 200), 'double'),
 ('hrir_r', (25, 50, 200), 'double')]
```

An example HRIR plot, similar to Figure 3, is shown in Figure 14.

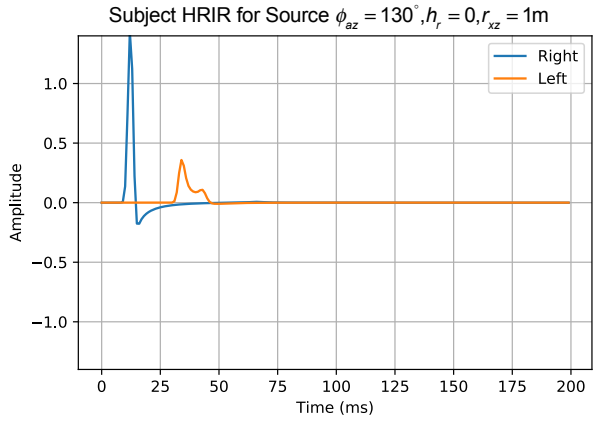


Fig. 14: *Example right/left HRIR plots for a particular arrival angle pulled from the CIPIC-like database entry created for a radius 8.75 cm sphere.*

Casual listening tests with a coarse fit human subject from CIPIC and the simple spherical model, indicate both similarities and differences. Coarse localization is similar between the two, but the spherical model seems *sterile*, that is the sound seems unnatural. The fact that coarse localization is present is an indication that the database is correct. Additional testing is planned.

Conclusions and Future Work

Development of the real-time signal processing aspect of the 3D audio simulator was a relatively simple task. This is a perfect application for the `pyaudio_helper` code module of `scikit-dsp-comm`. Working through the details of the coordinate transformations, and gain and parallax corrections on the geometry side, was a more complex undertaking. Likewise, working with the spherical head model calculations, first in the frequency domain (HRIR), and then the time domain (HRIR), was the most complex. The fact that recursions can be used to evaluate the needed special functions for sound pressure on the surface of a sphere, makes the generation of a CIPIC-like database entry take only a few minutes of compute time.

Informal testing of human subjects has gone well. Precise localization experiments using the static app have not been attempted just yet, as a formal pool human subjects has yet to be gathered. The virtual reality aspects of the dynamic app have received many positive comments from informal testing with those interested in 3D audio.

For future research, this simulator will be used to evaluate personalized HRIR fitting to human subjects, based on their upper torso anthropometrics. For the case of the spherical head, it is of interest to consider alternative HRIR grids. The 1 m radius 1250 point grid of angle pairs is no longer a limitation. For close range sound localization a different grid of angle pairs and with $r < 1$ m, can be used. This would make filter switching on the real-time DSP side of things finer grained, and hence more natural.

The Jupyter notebooks used in the analysis and development of this paper can be found on GitHub [3D_Audio]. This will give open access to anyone interested in trying out the simulator.

References

[HCI] *3-D Audio for Human/Computer Interaction*, (2019, June 30). Retrieved June 30, 2019, from https://www.ece.ucdavis.edu/cipic/spatial-sound/tutorial.

[CIPIC] *The CIPIC Interface Laboratory Home Page*, (2019, May 22). Retrieved May 22, 2019, from https://www.ece.ucdavis.edu/cipic.

[CIPICHRTF] *The CIPIC HRTF Database*, (2019, May 22). Retrieved May 22, 2019, from https://www.ece.ucdavis.edu/cipic/spatial-sound/hrtf-data.

[Fitzpatrick] Fitzpatrick, W., Wickert, M., and Semwal, S. (2013) 3D Sound Imaging with Head Tracking, *Proceedings IEEE 15th Digital Signal Processing Workshop/7th Signal Processing Education Workshop*.

[Wickert] *Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets*, (2018, July 15). Retrieved May 22, 2019, from DOI 10.25080/Majora-4af1f417-00e.

[ScipySignal] *Signal processing (scipy.signal)*, (2019, May 22). Retrieved May 22, 2019, from https://docs.scipy.org/doc/scipy/reference/signal.html.

[KEMAR] GRAS Sound & Vibration A/S, Head & Torso Simulators, from http://www.gras.dk/products/head-torso-simulators-kemar.

[Beranek] Beranek, L. and Mellow, T (2012). *Acoustics: Sound Fields and Transducers*. London: Elsevier.

[Duda] Duda, R. and Martens, W. (1998). Range dependence of the response of a spherical head model, *J. Acoust. Soc. Am. 104 (5)*.

[Bogelein] Bogelein, S., Brinkmann, F., Ackermann, D., and Weinzierl, S. (2018). Localization Cues of a Spherical Head Model. *DAGA Conference 2018 Munich*.

[3D_Audio] 3D audio simulator, (2019, June 16): Retrieved June 16, 2019, from https://github.com/mwickert/3D_Audio_Simulator.

An intelligent shopping list based on the application of partitioning and machine learning algorithms

Nadia Tahiri[§*], Bogdan Mazoure[‡], Vladimir Makarenkov[§]

Abstract—A grocery list is an integral part of the shopping experience of many consumers. Several mobile retail studies of grocery apps indicate that potential customers place the highest priority on features that help them to create and manage personalized shopping lists. First, we propose a new machine learning model written in Python 3 that predicts which grocery products the consumer will buy again or will try to buy for the first time, and in which store(s) the purchase will be made. Second, we introduce a smart shopping template to provide consumers with a personalized weekly shopping list based on their shopping history and known preferences. As the explanatory variables, we used available grocery shopping history, weekly product promotion information for a given region, as well as the product price statistics.

Index Terms—Machine Learning, Prediction, Long short-term memory, Convolutional Neural Network, Gradient Tree Boosting, F_1, Python, Sklearn, Tensorflow

Introduction

A typical grocery retailer offers consumers thousands of promotions every week to attract more consumers and thus improve its economic performance [TTR16]. The studies by Walters and Jamil (2002, 2003) ([WJ02] and [WJ03]) report that about 39% of all items purchased during a grocery shopping are weekly specials, and about 30% of consumers surveyed are very sensitive to the product prices, buying more promotional items than regular ones. With the recent expansion of machine learning methods, including deep learning, it seems appropriate to develop a series of methods that allow retailers to offer consumers attractive and cost-effective shopping baskets, as well as to offer tools to create smart personalized weekly shopping lists based on the purchase history, known preferences, and weekly specials available in local stores.

A grocery list is an integral part of the shopping experience of many consumers. Such lists serve, for example, as a reminder, a budgeting tool, or an effective way to organize weekly grocery shopping. In addition, several mobile retail studies indicate that potential customers place the highest priority on features that help them to create and manage personalized shopping lists interactively [NPS03] and [SZA16].

* *Corresponding author: tahiri.nadia@courrier.uqam.ca*
§ *Département d'Informatique, Université du Québec à Montréal, Case postale 8888, Succursale Centre-ville, H3C 3P8 Montréal, Canada*
‡ *Montreal Institute for Learning Algorithms (MILA) and McGill University,Montréal, Canada*

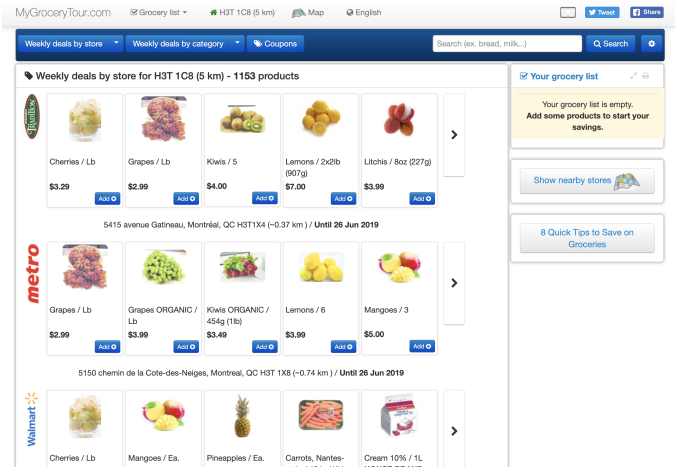


Fig. 1: *Screenshot of the MyGroceryTour.ca website for the postal code H3T 1C8 in Montreal. The website has been created to test our new machine learning model. It has been written in JavaScript, Bootstrap, and Python.*

Problem statement and proposal

In this section, we present the problem statement and describe the considered machine learning architecture. First, by using a Canadian grocery shopping database *MyGroceryTour.ca*[1] (see Figure 1), we partitioned consumers into classes based on their purchase histories. Then, this classification was used at the prediction stage. Since the real consumer data contained thousands of individual articles, we regrouped the products by their categories. A principal component analysis (linear and polynomial PCA [Jol11]) was first carried out to visualize the raw data and select the number of the main components to use when partitioning consumers into classes. The application of efficient partitioning methods, such as K-means [Jai10] and X-means [PM+00], allowed us to determine the number of classes of consumers, as well as their distribution by class. We used the Calinski-Harabazs cluster validity index [CH74] to determine the number of cluster in K-means. The Silhouette index [RPJ87] could be also used for this purpose.

Second, we developed a statistical model to predict which products previously purchased by a given consumer will be present in his/her next order. By using explanatory variables, such as available grocery shopping histories, information on the current promotions in stores of a given region, and commodity price

1. MyGroceryTour.ca

statistics, we developed a machine learning model which is able to:

i. Predict which groceries the consumer will want to buy again or will try to buy for the first time, as well as in which store(s) (within the area they usually shop in) the purchase(s) will be made;
ii. Create a smart shopping list by providing the consumer with a weekly shopping list, customized based on his/her purchase history and known preferences.

This list also includes recommendations regarding the optimal quantity of every product suggested. We also calculate the consumer's optimal weekly commute using the generalized travelling salesman algorithm (see Figure 2).

An F_1 statistics maximization algorithm [NCLC12] (see the Statistics section), based on dynamic programming, was used to achieve the objective (i). This will be of major interest to retailers and distributors. A deep learning method [GBC16], based on Recurrent Neural Networks (RNN) and Convolutional Neural Network (CNN), both implemented using the TensorFlow library [HLYX18], was used to achieve the objective (ii). Those implementations can provide significant benefits to consumers.

Our prediction problem can be reformulated as a binary prediction task. Given a consumer, the history of his/her previous purchases and a product with its price history, predict whether or not this product will be included in the grocery list of the consumer. Our approach applies a generative model to process the existing data, i.e., first-level models, and then uses the internal representations of these models as features of the second-level models. RNNs and CNNs were used at the first learning level and forward propagation neural networks (Feed-forward NN) was used at the second learning level.

Thus, depending on the user's u and the user's purchase history ($shop_{t-h:t}$, $h > 0$), we predict the probability that the product i is included in the current shopping basket $_{t+1}$ of u.

Dataset

In this section, we discuss the details of our synthetic and real datasets, the latter obtained from our website *MyGroceryTour.ca*.

Features

To perform the prediction only the features we found to be significant, such as *distance*, *special* rate, *products*, and *store*, were considered. All features used in our study are presented below:

- **user_id**: the user ID. We anonymized all data used in our study. $user\_id \in \underbrace{\{1 \cdots 374\}}_{\text{reals}} \cup \underbrace{\{375 \cdots 1,374\}}_{\text{generated}}$
- **order_id**: unique number of the basket. $order\_id \in \mathbb{Z}$
- **store_id**: unique number of the store. $store\_id \in \{1 \cdots 10\}$
- **distance**: distance to the store. $distance \in \mathbb{R}^+$
- **product_id**: unique number of the product. $product\_id = 49,684$. We tested our model with 1,000 products only (out of 49,684 products), which belonged to 5 out of the 24 available categories, i.e. *Fruits-Vegetables*, *Pasta-Flour*, *Organic Food*, *Beverages*, and *Breakfast*; the rest of the categories were not considered in our tests.
- **category_id**: unique category number for a product. $category\_id \in \{1 \cdots 24\}$
- **reorder**: the reorder is equal to 1 if the product has been ordered by this user in the past, 0 else. $reorders \in \{0,1\}$
- **special**: discount percentage applied to the product price at the time of purchase. $special \in \{[0\%, 15\%[, [15\%, 30\%[, [30\%, 50\%[, [50\%, 100\%[\}$

In total, we processed the data of 1374 users (i.e., consumers). Among them, we had 374 real users and 1000 users whose behaviour was generated following the distribution of real users (see Figure 3) and the consumer statistics available in the report by Statistics Canada (2017). The product categories were available for each product. So, the product category was one of the explanatory variables used in the model. In total, we considered 5 (of 24) product categories. The current version of our model does not allow a new product to be bought by the user (i.e., every user can only buy products that were present in at least one of its previous shopping baskets). The user IDs were not sequential because we only considered real users having a sufficient number of previous shopping baskets available (>50 baskets). The average basket size was also used to predict the content of the current basket size for each user.

Two types of features, categorical and quantitative variables, were present in our data. Only the *distance* and *special* features were quantitative variables, the rest of them were categorical. To manage the categorical variables, we applied a hashing scheme to deal with large scale categorical features. The hash function takes into account the input and output vector length. We used the *LabelEncoder* function of the *scikit-learn* package of Python (version 3).

Consumer profile

According to Statistics Canada there exist 3 consumer profiles (see [WJ03], [WJ02], and [TNTK16]). The first profile represents consumers who buy only promotional items. The second profile represents consumers who always buy the same products (without considering promotions). Finally, the third profile represents consumers who buy products whether they are in special or not. On our model, we plan to consider this information and make the prediction more personalized with respect to the consumer's profile.

Data Synthesis

Since the real dataset was not large enough to apply the appropriate machine learning methods, its size was increased by adding simulated data following the distribution of real data. The original dataset was composed of 374 users. It may be not enough to apply an appropriate machine learning method, and 1000 simulated users were added to our dataset. Thus, 72.7% of our data were simulated (1000 out of 1374 user histories were simulated). Here, we describe the simulated part of our dataset, and present in detail the results of the simulation step. For $store\_id$, we started with an initial store and changed stores based on the proportion of common products between baskets. If we assume that the store coordinates are normally and independently distributed $\mathcal{N}(0, \sigma^2)$, the distance between this store and the consumer home located at the origin $(0,0)$ follows a Rayleigh distribution [KR05] with the σ parameter. Finally, we increased the value of the *special* random variable. Its value has been drawn from a Boltzmann distribution [AAR+18]. We made sure that the generated baskets followed the same distribution that the original basket in terms of the basket size (see Figure 3).

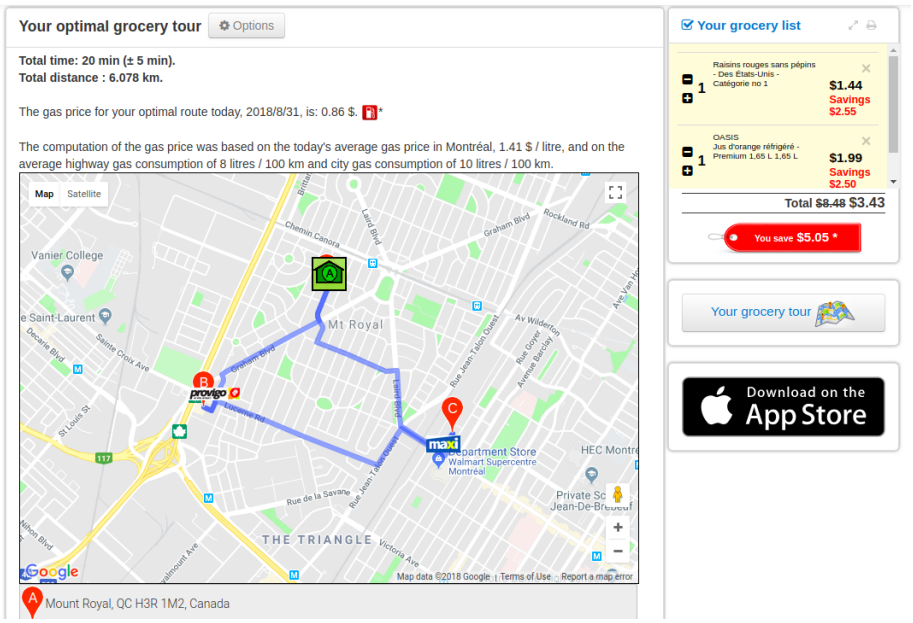


Fig. 2: *Screenshot of the MyGroceryTour.ca website displaying an optimal shopping journey calculated using the generalized travelling salesman algorithm.*

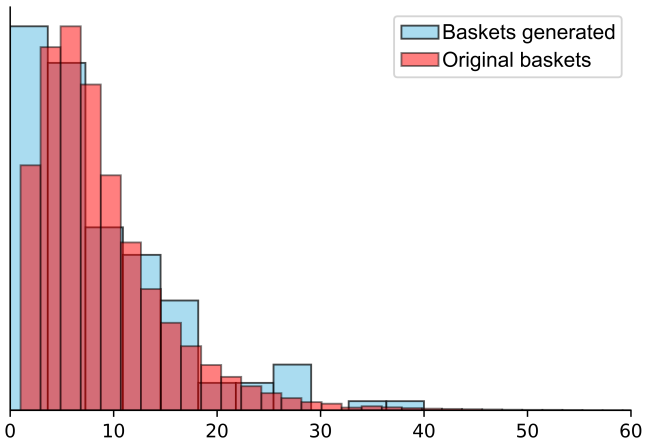


Fig. 3: *Difference in the basket size distribution between* ***Baskets generated*** *in blue and* ***Original baskets*** *in red.*

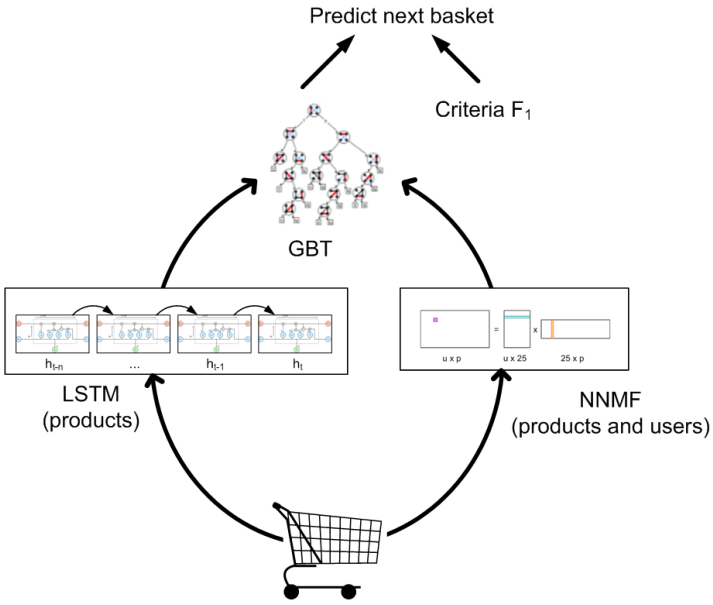


Fig. 4: *The graphical illustration of the proposed model intended to predict the content of the current grocery basket. At the first level of the model the LSTM and NNMF networks were used. At the second level of the model, the GBT model was applied. Finally, at the last step we predicted the current grocery basket using* F_1.

Preprocessing dataset

Initially, the data were saved in CSV files and stored in a MySQL database taking 1.4 GB of disk space. Then, the data were organized in a dataframe and processed using our Python script. We launched the preprocessing data tasks on the servers of Compute Canada. This step was carried out using 172 nodes and 40 cores with an Intel Gold 6148 Skylake CPU(2.4 GHz), and NVidia V100SXM2(16 GB of memory). We preprocessed the user data, the product data, and the department data. The preprocessing had a 48 hour limit and used 32 GB of the RAM memory.

Models

In this section, we present the workflow (see Figure 4) and the models we used. The graphical representation of the workflow in Figure 4 allowing one to predict the current consumer's basket using the three following models: LSTM, NNMF, and GBT (see the next section).

Long short-term memory (LSTM) network

The LSTM [HS97] is a recurrent neural network (RNN) that has an input, a hidden memory block, and an output layer. The memory block contains 3 gate units namely the input, forget, and output with a self-recurrent connection neuron [HS97].

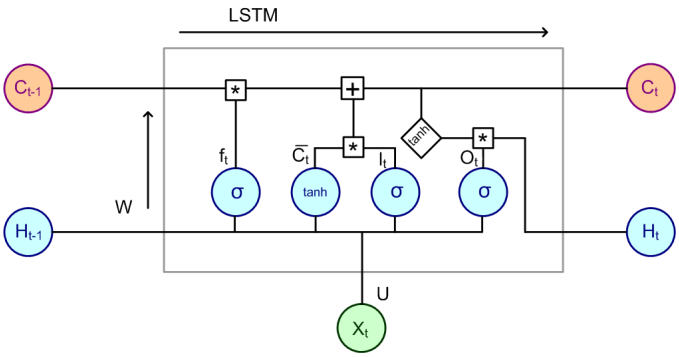


Fig. 5: *This figure shows a chain-structured LSTM network. An LSTM architecture contains the forget, learn, remember, and uses gates that determine the importance of the input data. In the LSTM unit represented in this figure, there are four different functions: sigmoid (σ), hyperbolic tangent (tanh), multiplication ($*$), and sum ($+$), making it easier to update the weights during the backpropagation process. Here X_t denotes the input vector, H_{t-1} is the previous cell output, C_{t-1} is the previous cell memory, H_t is the current cell output, C_t is the current cell memory. f_t is the forget gate with sigmoid function sigma, $\overline{C}_t$ and I_t corresponds to the input gate with tanh function, and finally O_t is the output gate with sigma function.*

- **Input gate** learns what information is to be stored in the memory block.
- **Forget gate** learns how much information from the memory block should be retained or forgotten.
- **Output gate** learns when the stored information can be used.

Figure 5 illustrates the proposed architecture and summarizes the details of our network model.

A combined RNN and CNN network was trained to predict the probability that a given user will order a given product at each timestep. A timestep was defined by the composition of the basket and the store location on the map (see Figure 2). Here, RNN was a single-layer LSTM and CNN was a 6-layer causal CNN with dilated convolutions. The width of the CNN was equal to 1374 (i.e., the number of users), the height was equal to 8 (i.e., the number of features), and the depth was equal to 100 (i.e., the number of orders). The last layer was a fully-connected layer that was making the final classification. The CNN network was used as a feature extractor and the LSTM network as a sequential learner.

Overall characteristics of the neural networks used in our project are as follow:

```
nn = rnn(
 reader=dr,
 log_dir=os.path.join(base_dir,
                      'logs'),
 checkpoint_dir=os.path.join(base_dir,
                             'checkpoints'),
 prediction_dir=os.path.join(base_dir,
                             'predictions'),
 optimizer='adam',
 learning_rate=.001,
 lstm_size=512,
 batch_size=64,
 num_training_steps=300,
 early_stopping_steps=10,
 warm_start_init_step=0,
 regularization_constant=0.0,
 keep_prob=1.0,
 enable_parameter_averaging=False,
 num_restarts=2,
 min_steps_to_checkpoint=100,
 log_interval=20,
 num_validation_batches=4,
)
```

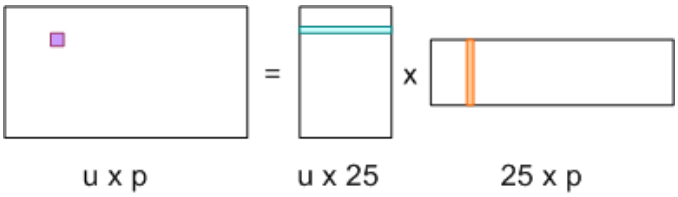


Fig. 6: *Decomposition of the user_id (u in Figure) by product_id (p in Figure). The first matrix accounts for the products purchased by the user (i.e. order count), whereas the second and the third matrices account for the representations of the user and the product, respectively.*

We considered the *Adam* optimizer which is a good default implementation of gradient descent. The learning rate was equal to 0.001 to control how long the weights should be updated in response to the estimated gradient at the end of each batch. The size of the hidden state of an LSTM unit was fixed to 512. Batch size corresponds to the number of samples between updates of the model weights. It was set to 64 during the training process. Also, we set to 4 the number of validation batches. The Tensorflow package was used to implement our *rnn* class that account for the features described in the previous section. The *rnn* class structure was organized using the four following functions: 1) constructor function, 2) loss score function calculation, 3) getter function, and 4) output score function.

```
import TFBaseModel

class rnn(TFBaseModel):
  def __init__(self,
               lstm_size,
               dilations,
               filter_widths,
               skip_channels,
               residual_channels,
               **kwargs):
    ...
  def calculate_loss(self):
    ...
  def get_input_sequences(self):
    ...
  def calculate_outputs(self, x):
    ...
```

Non-negative matrix factorization (NNMF) network

Non-negative matrix factorization NNMF [LS01] is a series of algorithms in multivariate analysis and linear algebra in which a matrix X is factorized into two matrices W and H having the property that all three matrices have no negative elements. This non-negativity makes the resulting matrices easier to utilize. We factorize the matrix X (i.e. matrix of *user_id* by *product_id*) into two matrices W (i.e. *user_id*) and H (i.e. *product_id*), so that the matrix representation can be formulated as: $X \approx WH$ (see Figure 6).

NNMF is a powerful machine learning method. It has been proved that NNMF converse to at least a locally optimal solution [LS01]. NNMF is trained on the matrix of the *user*product* counts.

Gradient Boosted Tree (GBT) network

GBT [Fri02] is an iterative algorithm that combines simple parameterized functions with low performance (i.e. high prediction error) to produce a highly accurate prediction rule. GBT utilizes an ensemble of weak learners to boost performance; this makes it a good candidate model for predicting the grocery shopping list. It requires little data preprocessing and tuning of parameters while yielding interpretable results, with the help of partial dependency plots and other investigative tools. Further, GBT can model complex interactions in a simple recommendation system and be applied in both classification and regression with a variety of response distributions, including Gaussian [Car03], Bernoulli [CMW16], Poisson [PJ73], and Laplace [Tay19]. The composition of the shopping history list is not complete in the sense that we do not have the composition of the baskets for each user for all weeks. Finally, missing values in the collected data can be easily managed.

The data were divided into two groups (training and validation sets) which comprised 90% and 10% of the data, respectively. After simulating the dataset, the strategy used was to merge real and simulated data and then split them into two groups (training and validation sets). The test set was composed of both real and simulated data. The final model included two neural networks and a GBT classifier. Once trained, it was used to predict in "real time" the content of the current grocery basket, based on the history of purchases and the current promotions in neighbouring stores. In fact, the "real time" does not mean "second-by-second", but rather "day-by-day". We scheduled it using the crontab tool. Based on the validation loss function, we removed the following parameters from our input data: 1) LSTM Category and 2) LSTM size of the next basket.

The last layer included a GTB classifier used to predict the products that will be purchased during the current week. GBT model was modelled using "by user" and "by order" frameworks. The classifier contained two classes: 0 (i.e. the product will be bought) and 1 (i.e. the product won't be bought).

First level model (feature extraction)

Our goal was to find a diverse set of representations using neural networks (see Table 1). Table 1 summarizes the top-level models used by the algorithm. We described each type of model used for every representation (e.g. *Products*, *Category*, *Size of the basket*, and *Users*). We estimated the probability of the $product_i$ to be included into the next basket $order_{t+1}$ with $orders_{t-h}$, where t represents the current time, $t+1$ represents the next time, and $t-h$ represents all previous time periods (i.e. time history). We decomposed the matrix {user,product} into two matrices, one corresponding to the user and another to the product. We predicted the probability to have the $product_i$ in the next $order_{t+1}$, taking into account the purchase history of the current user. We used an LSTM network with 300 neurons. Finally, we optimized the size of the next order by minimizing the root mean square error (RMSE).

Latent representations of entities (embeddings)

For each $a \in \mathscr{A}$, an embedding $T : \mathscr{A} \to \mathbb{R}^d$ returns a vector d-dimensionel. If $\mathscr{A} \subset \mathbb{Z}$, T is a matrix $|\mathscr{A}| \times d$ learned by backpropagation. We represented in Table 2 all dimensions of each model used.

Representation	Description	Type
Products	Predict P(product$_i \in$ order$_{t+1}$) with orders$_{t-h,t}$, $h > 0$.	LSTM (300 neurons)
Categories	Predict P($\exists i$: product$_{i,t+1} \in$ category$_r$).	LSTM (300 neurons)
Size	Predict the size of the order$_{t+1}$.	LSTM (300 neurons)
Users Products	Decomposed $V_{(u\times p)} = W_{(u\times d)} H^T_{(p\times d)}$	Dense (50 neurons)

TABLE 1: *Top-level models used. The figure shows the representation, the description, and the type of products, the categories, the size of baskets, and the matrix users/products.*

Model	Embedding	Dimensions
LSTM Products	Products	$49,684 \times 300$
LSTM Products	Categories	24×50
LSTM Products	Categories	$50 \to 10$
LSTM Products	Users	$1,374 \times 300$
NNMF	Users	$1,374 \times 25$
NNMF	Products	$49,684 \times 25$

TABLE 2: *Dimensions of the representations learned by different models at the first level of the model.*

Second level model: Composition of baskets

The resulting basket was chosen according to the final reorganization probabilities, selecting the subset of products with the expected maximum F_1 score, see [LEN14] and [NCLC12]. This score is frequently used when the relevant elements are scarce.

$$\max_{\mathscr{P}} \mathbb{E}_{p' \in \mathscr{P}}[F_1(\mathscr{P})] = \max_{\mathscr{P}} \mathbb{E}_{p' \in \mathscr{P}} \left[\frac{2\sum_{i \in \mathscr{P}} \mathrm{TP}(i)}{\sum_{i \in \mathscr{P}} (2\mathrm{VP}(i) + \mathrm{FN}(i) + \mathrm{FP}(i))} \right],$$

where True Positive $(TP) = \mathbb{I}[\lfloor p(i) \rceil = 1]\mathbb{I}[R_i = 1]$, False Negative $(FN) = \mathbb{I}[\lfloor p(i) \rceil = 0]\mathbb{I}[R_i = 1]$, False Positive $(FP) = \mathbb{I}[\lfloor p(i) \rceil = 1]\mathbb{I}[R_i = 0]$ and $R_i = 1$ if the product i was bought in the basket $p' \in \mathscr{P}$, else 0.\ We used $\mathbb{E}_X[F_1(Y)] = \sum_{x \in X} F_1(Y = y|x)P(X = x)$

Statistics

Here, we present the results obtained using the proposed method. The *F-measure* (see Equation 1) metric was used to evaluate the performance of the method.

Statistical score

F-measure, or F_1, is a well-known and reliable evaluation statistic (see [JOA05]). The F_1 value of 1 means perfect accuracy.

$$F - measure = F1 = \frac{2 \times Precision \times Recall}{(Precision + Recall)} \tag{1}$$

Python Script

The final reorder probabilities were computed as the weighted average of the outputs from the second-level models. The final basket was chosen by using these probabilities and selecting the product subset with a maximum expected F1-score. In our implementation, we used *f1_optimizer* implemented in **F1Optimizer** package. The implementation of [NCLC12] is available in [F1Optimizer]. The select_products function in Python script was the following:

```
from f1_optimizer import F1Optimizer

def select_products(x):
 series = pd.Series()

```

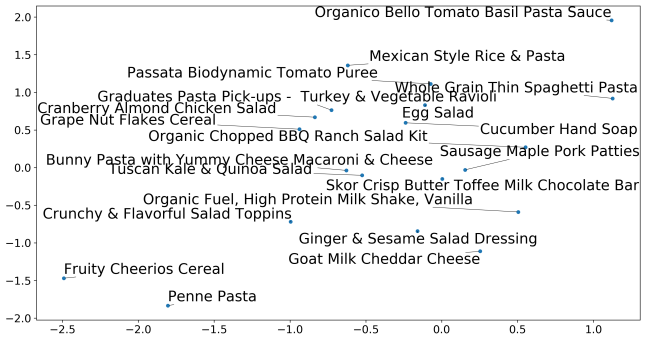


Fig. 7: *Embeddings of 20 random products projected into 2 dimensions.*

```
for prod in x['product_id'][x['label'] > 0.5:
  if prod != 0:
   true_products = [str(prod)].values]
  else:
   true_products = ['None'].values]

if true_products:
 true_products = ' '.join(true_products)
else:
 true_products = 'None'

prod_preds_dict = dict(zip(x['product_id'].values,
                           x['prediction'].values))
none_prob = prod_preds_dict.get(0, None)
del prod_preds_dict[0]

other_products = np.array(prod_preds_dict.keys())
other_probs = np.array(prod_preds_dict.values())

idx = np.argsort(-1*other_probs)
other_products = other_products[idx]
other_probs = other_probs[idx]

opt = F1Optimizer.max_expectation(other_probs,
                                  none_prob)

best_prediction = ['None'] if opt[1] else []
best_prediction += list(other_products[:opt[0]])

if best_prediction:
 predicted_products = ' '.join(map(str,
                               best_prediction))
else:
 predicted_products = 'None'

series['products'] = predicted_products
series['true_products'] = true_products

return true_products, predicted_products, opt[-1]
```

Results

Figure 7 illustrates PCA of 20 random products projected into 2 dimensions. These results show clearly the presence of the cluster of products, including the Pasta sauce and Pasta group articles. This embedding plot was generated with 20 random products. Some trends can be observed here, but there are also some exceptions, as it often happens with real data. In Table 2, Pasta Group was included into the product Categories. In fact, this result can help identify the consumer buying behaviour.

PCA was performed to visualize the clustering of 20 selected products. It was used to show that some products are frequently bought together with the other products. Such a clustering was not used explicitly in our model, by an artificial network model is supposed to capture and take it into account implicitly in order to provide a better prediction. F_1 in Figure 8 (a) shows that the profiles of all promotions are similar. In the perspective, it would be interesting to include in our model the product weight based on some additional available statistics. For example, according to Statistics Canada - 2017, only 5% of all specials had a rebate of 50% and larger, whereas 95% of them had a smaller rebate. The use of theses weights could make the model more robust.

Figure 8 (b) indicates that all stores follow similar profiles in our model.

This plot presents the distribution of the F1-score results with respect to the promotions and stores. We can observe that the distributions of the promotions and stores are very similar. Finally, this plot suggests the absence of the bias for these two model parameters. Figure 9 and Table 3 report the values of the F_1 metric for the products whose inclusion into the consumer's basket was either very easy or very hard to predict. The first group of products includes the articles of restriction regimes such as *diet cranberry fruit juice*, *purified water*, and *total 0% blueberry acai greek yogurt*.

Table 3 presents only the products with the five highest and the five lowest values of *F_1* (the average, in this case, was taken over all users who purchased these products).

We also evaluated the prediction quality of our model (see Section 'Statistic scores') using the *sklearn* metrics (see below):

```
from sklearn.metrics import make_scorer,
                            accuracy_score,
                            f1_score,
                            recall_score
```

The results reported in Table 5 suggest that a better model accuracy was obtained when the original dataset of 374 real users was enriched by 1,000 artificial users. The accuracy of 49% was

Product	F_1
Gogo Squeez Organic Apple Strawberry Applesauce	0.042057
Organic AppleBerry Applesauce on the Go	0.042057
Carrot And Celery Sticks	0.042057
Gluten Free Peanut Butter Berry Chewy	0.042057
Organic Italian Balsamic Vinegar	0.049325
Diet Cranberry Fruit Juice	0.599472
Purified Water	0.599472
Vanilla Chocolate Peanut Butter Ice Cream Bars	0.599472
Total 0% with Honey Nonfat Greek Strained Yogurt	0.590824
Total 0% Blueberry Acai Greek Yogurt	0.590824

TABLE 3: *The average value of F_1 for all products considered.*

Product	Number of baskets
Banana	6138
Strawberries	3663
Organic Baby Spinach	1683
Limes	1485
Cantaloupe	1089
Bing Cherries	891
Small Hass Avocado	891
Organic Whole Milk	891
Large Lemon	792
Sparkling Water Grapefruit	792

TABLE 4: *The 10 most popular products included in the predicted baskets. The top products were taken from a subset comprising 2% of all available products.*

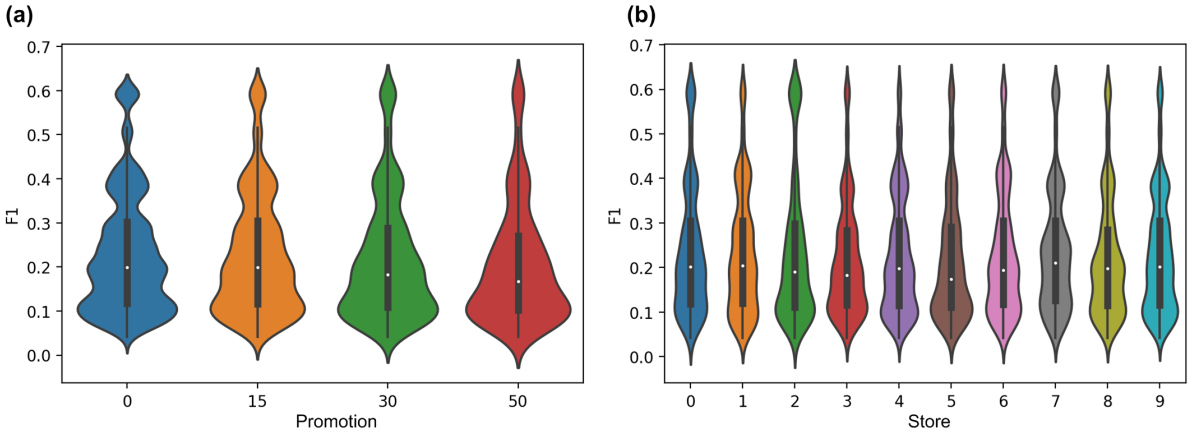


Fig. 8: *Distribution of F_1 measures against rebates (a), and stores (b).*

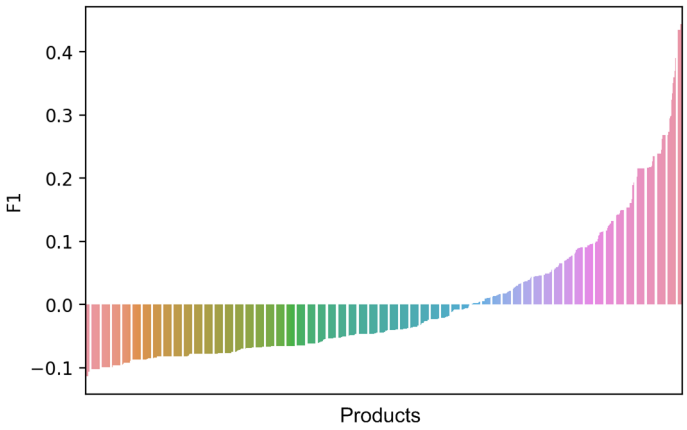


Fig. 9: *Distribution of the F_1 scores relative to the products around average.*

Statistics score	Real data	Real and augmented data
Accuracy	0.27	0.49
Precision	0.27	0.49
Recall	0.51	0.7
F-measure	0.22	0.37

TABLE 5: *Statistical scores results obtained for real data, and for real + artificial augmented data. The table clarifies the impact of using augmented data instead in addition to the real ones.*

obtained for the augmented dataset, compared to the accuracy of 27% for the original dataset.

Conclusions and Future Work

We analyzed grocery shopping data generated by the users of the site *MyGroceryTour.ca*. We developed a new machine learning model to predict which grocery products the consumer will buy and in which store(s) of the region he/she will do grocery shopping. We created an intelligent shopping list based on the shopping history of each consumer and his/her known shopping preferences. The originality of the approach, compared to the existing methods, is that in addition to the purchase history we also considered the promotions available, possible purchases in different stores, and the distance between these stores and the consumer's home.

We have modelled the habits of the MyGroceryTour.ca site consumers using deep neural networks. Two types of neural networks were applied at the learning stage: Recurrent neural networks (RNN) and Forward-propagating neural networks (Feed-forward NN). The value of the F_1 statistic that represents the quality of the model needs could be increased in the future by considering additional explanatory features and product weights. The constant influx of new data on *MyGroceryTour* will allow us to improve the model's results.

In the future, we plan to predict the grocery store that will be visited next, and include the recommended product quantities in the basket proposed to the user.

Acknowledgments

The authors thank the members of PyCon Canada for their valuable comments on this project. We would like to thanks SciPy conference, Dillon Niederhut, David Shupe, Chris Calloway, and anonymous reviewers for their valuable comments on this manuscript. This work used resources of Compute Canada. This work was supported by Natural Sciences and Engineering Research Council of Canada and Fonds de Recherche sur la Nature et Technologies of Quebec.

Abbreviations

- CNN - Convolutional Neural Network
- GBT - Gradient Tree Boosting
- LSTM - Long Short-Term Memory
- ML - Machine Learning
- NN - Neural Networks
- NNMF - Non-Negative Matrix Factorization
- PCA - Principal Component Analysis
- RMSE - Root Mean Square Error
- RNN - Recurrent Neural Networks

References

[AAR+18] Amin, Mohammad H., Evgeny Andriyash, Jason Rolfe, Bohdan Kulchytskyy, and Roger Melko. *Quantum boltzmann machine*. Physical Review X, 8(2):021050, 2018. DOI: https://doi.org/10.1103/PhysRevX.8.021050

[Car03] Rasmussen, Carl Edward. *Gaussian processes in machine learning*. In Summer School on Machine Learning, pp. 63:71. Springer, Berlin, Heidelberg, 2003. DOI: https://doi.org/10.1007/978-3-540-28650-9_4

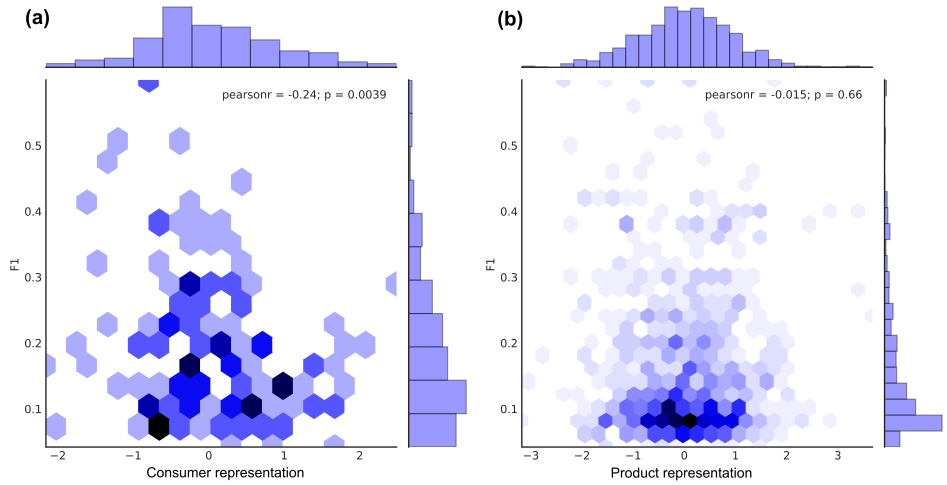


Fig. 10: *Distribution of the F_1 scores with respect to the consumers and products.*

[CH74] Caliński, T. and Harabasz, J., 1974. *A dendrite method for cluster analysis*. Communications in Statistics-theory and Methods, 3(1), pp.1-27. DOI: https://doi.org/10.1080/03610917408548446

[CMW16] Maddison, Chris J., Andriy Mnih, and Yee Whye Teh. *The concrete distribution: A continuous relaxation of discrete random variables*. arXiv preprint arXiv:1611.00712, 2016. https://arxiv.org/pdf/1611.00712.pdf

[F1Optimizer] Kaggle post, *F1-Score Expectation Maximization in O(n²)*, 2017. https://www.kaggle.com/mmueller/f1-score-expectation-maximization-in-o-n

[Fri02] Jerome H. Friedman. *Stochastic gradient boosting*. Computational Statistics & Data Analysis, 38(4):367–378, 2002. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[HLYX18] Hao, L., Liang, S., Ye, J. and Xu, Z., 2018. *TensorD: A tensor decomposition library in TensorFlow*. Neurocomputing, 318, pp. 196-200. DOI: https://doi.org/10.1016/j.neucom.2018.08.055

[HS97] Sepp Hochreiter and Jurgen Schmidhuber. *Long short-term memory*. Neural computation, 9(8):1735–1780, 1997. DOI: https://doi.org/10.1162/neco.1997.9.8.1735

[Jai10] Anil K Jain. *Data clustering: 50 years beyond k-means*. Pattern recognition letters, 31(8):651–666, 2010. DOI: https://doi.org/10.1016/j.patrec.2009.09.011

[JOA05] Joachims, T., 2005. *A support vector method for multivariate performance measures*. In Proceedings of the 22nd international conference on Machine learning (pp. 377-384). ACM. DOI: https://dl.acm.org/citation.cfm?doid=1102351.1102399

[Jol11] Ian Jolliffe. *Principal component analysis*. Springer, 2011. DOI: https://doi.org/10.1007/978-3-642-04898-2_455

[KR05] Debasis Kundu and Mohammad Z Raqab. *Generalized rayleigh distribution: different methods of estimations*. Computational statistics & data analysis, 49(1):187–200, 2005. DOI: https://doi.org/10.1016/j.csda.2004.05.008

[LEN14] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. *Optimal thresholding of classifiers to maximize f1 measure*. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 225–239. Springer, 2014. DOI: https://doi.org/10.1007/978-3-662-44851-9_15

[LS01] Lee, D.D. and Seung, H.S. *Algorithms for non-negative matrix factorization*. In Advances in neural information processing systems, pp. 556-562, 2001.

[NCLC12] Ye Nan, Kian Ming Chai, Wee Sun Lee, and Hai Leong Chieu. *Optimizing f-measure: A tale of two approaches*. arXiv preprint arXiv:1206.4625, 2012. https://arxiv.org/ftp/arxiv/papers/1206/1206.4625.pdf

[NPS03] Erica Newcomb, Toni Pashley, and John Stasko. *Mobile computing in the retail arena*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 337–344. ACM, 2003. DOI: https://doi.org/10.1145/642667.642670

[PJ73] Consul, Prem C., and Gaurav C. Jain. *A generalization of the Poisson distribution*. Technometrics 15(4):791-799, (1973).

[PM+00] Dan Pelleg, Andrew W Moore, et al. *X-means: extending kmeans with efficient estimation of the number of clusters*. In Icml, volume 1, pp. 727–734, 2000.

[RPJ87] Rousseeuw, P.J., 1987. *Silhouettes: a graphical aid to the interpretation and validation of cluster analysis*. Journal of computational and applied mathematics, 20, pp.53-65. DOI: https://doi.org/10.1016/0377-0427(87)90125-7

[SZA16] Szpiro, S., Zhao, Y. and Azenkot, S. *Finding a store, searching for a product: a study of daily challenges of low vision people*. In Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pp. 61-72. ACM, 2016. DOI: https://doi.org/10.1145/2971648.2971723

[Tay19] Taylor, James W. *Forecasting value at risk and expected shortfall using a semiparametric approach based on the asymmetric Laplace distribution*. Journal of Business & Economic Statistics 37(1):121-133, 2019. DOI: https://doi.org/10.1080/07350015.2017.1281815

[TNTK16] Arry Tanusondjaja, Magda Nenycz-Thiel, and Rachel Kennedy. *Understanding shopper transaction data: how to identify crosscategory purchasing patterns using the duplication coefficient*. International Journal of Market Research, 58(3):401–419, 2016. DOI: https://doi.org/10.2501/ijmr-2016-026

[TTR16] Arry Tanusondjaja, Giang Trinh, and Jenni Romaniuk. *Exploring the past behaviour of new brand buyers*. International Journal of Market Research, 58(5):733–747, 2016. DOI: https://doi.org/10.2501/ijmr-2016-042

[WJ02] Rockney Walters and Maqbul Jamil. *Measuring cross-category specials purchasing: theory, empirical results, and implications*. Journal of Market-Focused Management, 5(1):25–42, 2002.

[WJ03] Rockney G Walters and Maqbul Jamil. *Exploring the relationships between shopping trip type, purchases of products on promotion, and shopping basket profit*. Journal of Business Research, 56(1):17–29, 2003. DOI: https://doi.org/10.1016/s0148-2963(01)00201-6

Parameter Estimation Using the Python Package pymcmcstat

Paul R. Miles[‡], Ralph C. Smith[‡*]

Abstract—A Bayesian approach to solving inverse problems provides insight regarding model limitations as well as the underlying model and observation uncertainty. In this paper we introduce pymcmcstat, which provides a wide variety of tools for estimating unknown parameter distributions. For scientists and engineers familiar with least-squares optimization, this package provides a similar interface from which to expand their analysis to a Bayesian framework. This package has been utilized in a wide array of scientific and engineering problems, including radiation source localization and constitutive model development of smart material systems.

Index Terms—Markov Chain Monte Carlo (MCMC), Delayed Rejection Adaptive Metropolis (DRAM), Parameter Estimation, Bayesian Inference

Introduction

Many scientific problems require calibration of model parameters. This process typically involves comparing a model with a set of data, where the data either comes from experimental observations or high-fidelity simulations. The model parameters are calibrated in a manner such that the model fits the data; i.e., observations are used to inversely determine the model inputs that led to that output. A common example of this procedure is least-squares optimization, which is used in a wide variety of scientific disciplines. Least-squares and many other methods exist for solving these inverse problems, but an important question to ask is whether or not they account for the underlying uncertainty.

Uncertainty exists in all areas of scientific research and it arises for various reasons. A familiar source of uncertainty in data is simply a certain amount of random noise. Alternatively, uncertainty also occurs due to missing physics in the model or from lack of knowledge. Modeling scientific and engineering problems presents many challenges and often times requires compromise. No model ever fully captures the physics; however, the model may still be useful for different applications [Box76]. With that in mind, we now highlight an approach to inverse problems that helps address uncertainty in the development of scientific and engineering models.

To quantify the uncertainty in our modeling problem, we utilize Bayesian inference. The key point in this approach stems from the interpretation of the parameters within the model. A Bayesian approach treats these unknown model parameters as random variables; i.e., they have an underlying probability distribution that can be used to describe them. This contrasts a frequentist approach which assumes the parameters are unknown but have a fixed value. The goal of Bayesian model calibration is to infer the parameter distributions. This approach to inverse problems provides insight into model limitations as well as an accurate estimation of the underlying model and observation uncertainty. A brief summary is provided in the next section, and more details regarding these methods can be found elsewhere [Smi14].

The Python package pymcmcstat [Mil19b] provides a robust platform for performing Bayesian model calibration. Procedurally, the user provides data, defines model parameters and settings, and sets up simulation options. As many intended users may be unfamiliar with Bayesian methods, the default package behavior requires minimal knowledge of statistics. In fact, like many optimization problems, the user's main responsibility is to provide a sum-of-squares error function, which will become clear throughout the examples in this paper.

Within pymcmcstat, we use Markov Chain Monte Carlo (MCMC) methods to solve the Bayesian inverse problem [Smi14]. As many Python packages currently exist for performing MCMC simulations, we had several goals in developing this code. To our knowledge, no current package contains the n-stage delayed rejection algorithm, so pymcmcstat was intended to fill this gap. Delayed rejection may be an unfamiliar concept, so more details are provided in the discussion of Metropolis algorithms in a later section. Furthermore, many researchers in our community have extensive experience using the MATLAB toolbox mcmcstat[1]. Our implementation provides a similar user environment, while exploiting Python structures. We hope to decrease dependence on MATLAB in academic communities by advertising comparable tools in Python.

This package has been applied to a wide variety of engineering problems, including constitutive model development of smart material systems as well as radiation source localization. Several example problems will be presented later on, but first we will outline the package methodology.

Methodology

Knowledge of Bayesian statistics is important to understanding the theory, but it is not necessarily required information for using pymcmcstat. We provide a brief overview of the Bayesian

‡ *Department of Mathematics, North Carolina State University, Raleigh, NC 27695*
* *Corresponding author: rsmith@ncsu.edu*

1. https://mjlaine.github.io/mcmcstat/

approach and then explain the key terms that impact the user by going through a basic example.

Bayesian Framework

The goal of Bayesian inference is to estimate the posterior densities $\pi(q|F^{obs}(i))$, which quantify the probability of parameter values given a set of observations. From Bayes' relation

$$\pi(q|F^{obs}(i)) = \frac{\mathscr{L}(F^{obs}(i)|q)\pi_0(q)}{\int_{\mathbb{R}^p}\mathscr{L}(F^{obs}(i)|q)\pi_0(q)dq}, \tag{1}$$

we observe that the posterior is proportional to the likelihood and prior functions. The function $\mathscr{L}(F^{obs}(i)|q)$ describes the likelihood of the observations given a parameter set, and any information known *a priori* about the parameters is defined in the prior distribution $\pi_0(q)$. The denominator ensures that the posterior integrates to unity. Note, the integral in the denominator involves integrating over $\mathbb{R}^p$, where p is the number of model parameters.

The pymcmcstat package is designed to work with statistical models of the form

$$F^{obs}(i) = F(i;q) + \varepsilon_i, \text{ where } \varepsilon_i \sim N(0,\sigma^2).$$

We expect the observations $F^{obs}(i)$ (experimental data or high-fidelity simulations) to equal the model response $F(i;q)$ plus independent and identically distributed error ε_i with mean zero and observation error variance σ^2. A direct result of assuming a statistical model of this nature is that the likelihood function becomes

$$\mathscr{L}(F^{obs}(i)|q) = \exp\Big(-\frac{SS_q}{2\sigma^2}\Big), \tag{2}$$

where $SS_q = \sum_{i=1}^{N_{obs}}[F^{obs}(i) - F(i,q)]^2$ is the sum-of-squares error (N_{obs} is the number of data points). This is consistent with the observations being independent and identically distributed with $F^{obs}(i) \sim N(F(i;q),\sigma^2)$. As the observation error variance σ^2 is unknown in many cases, we will often include it as part of the inference process.

Direct evaluation of (1) is often computationally untenable due to the integral in the denominator. To avoid the issues that arise due to quadrature, we alternatively employ Markov Chain Monte Carlo (MCMC) methods. In MCMC, we use sampling based Metropolis algorithms [MRR+53] whose stationary distribution is the posterior density $\pi(q|F^{obs}(i))$. What this means is that we sample parameter values, evaluate the numerator of Bayes' equation (1), and accept or reject parameter values using a Metropolis algorithm. More details regarding Metropolis algorithms are provided in a later section.

Basic Example

At the end of the day, many users do not need to know the statistical background, but they can still appreciate the information gained from using the Bayesian approach. Below we outline the key components of pymcmcstat and explain their relationship to the Bayesian approach described above. Procedurally, to calibrate a model using pymcmcstat, the user will need to provide the following pieces:

1) Import and initialize MCMC object.
2) Add data to the simulation - $F^{obs}(i)$. These may be either experimental measurements or high-fidelity model results.
3) Define model function: The user needs to define a model of the form $F(i,q)$; i.e., a model that depends on a set of parameters q. Strictly speaking the model can be created in any language the user desires so long as it can be called within your Python script. For example, if your model code is written in C++ or Fortran, this is easily done using ctypes[2]. Note, the model does not need to be a separate `def` statement, but can be included directly in the sum-of-squares function.
4) Define sum-of-squares function - SS_q. The sum-of-squares error between the model and data will be used in evaluating the likelihood function $\mathscr{L}(F^{obs}(i)|q)$.
5) Define model settings and simulation options. More details regarding these features will be provided in subsequent sections.
6) Add model parameters - q. The user must specify the parameters in the model that need to be calibrated as well as define any limits regarding potential values those parameters can have. By defining parameter minimum and/or maximum limits, the user has specified the prior function $\pi_0(q)$. By default, pymcmcstat assumes a uniform distribution for all parameters; i.e., there is equal probability of the parameter being a particular value between the minimum and maximum limit.
7) Execute simulation.
8) Analyze parameter chains. The chains reflect the sampling history of the MCMC simulation.

Let's walk through a basic example to see how all these pieces work together. To start, we will generate some fictitious data,

```
import numpy as np
x = np.linspace(0, 1, num=100)
y = 2.0*x + 3.0 + 0.1*np.random.standard_normal(
                          x.shape)
```

Note, we assume data where observations `y` have been made at independent points `x`, which are uniformly distributed between 0 and 1. The observations follow a linear trend with slope 2 and offset 3. To make the data realistic we add random noise to the observations of the form $\varepsilon_i \sim N(0,\sigma^2)$. In this case we define the observation error standard deviation to be $\sigma = 0.1$.

In this case we know what the model should be because we used it to generate the data. We want to fit a linear model (i.e., $F(i,q=[m,b]) = mx_i + b$) to the observations. To calibrate this model with pymcmcstat, the basic implementation is as follows:

```
# import and initialize
from pymcmcstat.MCMC import MCMC
mcstat = MCMC()
# Add data
mcstat.data.add_data_set(x, y)
# Define sum of squares function
def ssfun(q, data):
    m, b = q  # slope and offset
    x = data.xdata[0]
    y = data.ydata[0]
    # Evaluate model
    ymodel = m*x + b
    res = ymodel - y
    return (res ** 2).sum(axis=0)
# Define model settings
mcstat.model_settings.define_model_settings(
    sos_function=ssfun)
# Define simulation options
mcstat.simulation_options.define_simulation_options(
    nsimu=10.0e3)  # No. of MCMC simulations
```

2. https://docs.python.org/3/library/ctypes

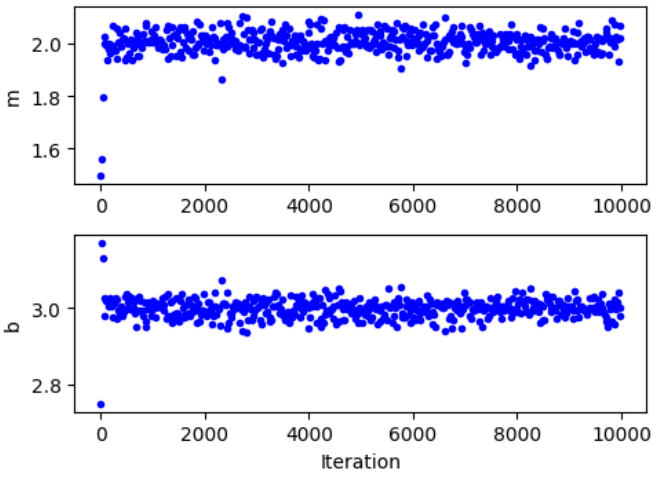


Fig. 1: *Parameter chains obtained with all 10,000 realizations of the linear model.*

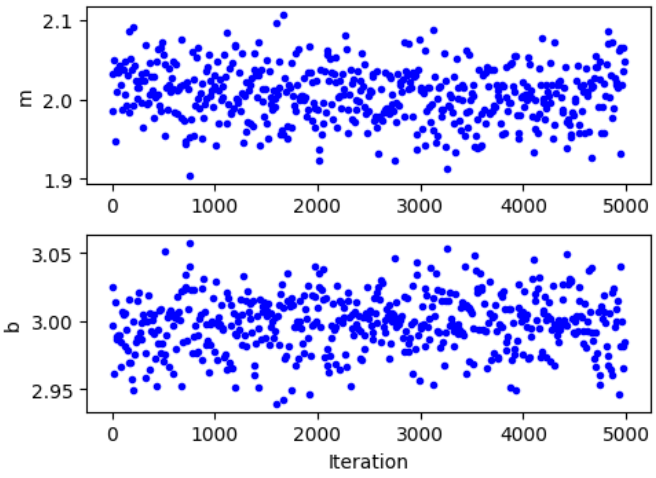


Fig. 2: *Parameter chains obtained with the final 5,000 realizations of the linear model.*

```
# Add model parameters
mcstat.parameters.add_model_parameter(
    name='m',
    theta0=2.)  # initial value
mcstat.parameters.add_model_parameter(
    name='b',
    theta0=2.75,  # initial value
    minimum=-5,  # lower limit
    maximum=5)  # upper limit
# Run simulation
mcstat.run_simulation()
```

We can check the results of the MCMC simulation by displaying the chain statistics. Note, we typically remove the first part of the sampling chain as it may not have converged to the correct posterior depending on the initial value.

```
# Extract results
results = mcstat.simulation_results.results
chain = results['chain']
burnin = int(chain.shape[0]/2)
# display chain statistics
mcstat.chainstats(chain[burnin:, :], results)
```

This will output to your display

```
name   : mean     std      MC_err   tau      geweke
m      : 2.0059   0.0348   0.0015   7.1351   0.9912
b      : 2.9983   0.0206   0.0009   7.9169   0.9962
```

Recall that the data was generated with a slope of 2 and offset of 3, so the algorithm appears to be converging to the correct values. Additional items displayed include normalized batch mean standard deviation (`MC_err`), autocorrelation time (`tau`), and Geweke's convergence diagnostic (`geweke`) [BR98].

A typical part of analyzing the results is to visualize the sampling history of the MCMC process. This is accomplished by using pymcmcstat's `plot_chain_panel` method.

```
mcpl = mcstat.mcmcplot  # initialize plotting methods
mcpl.plot_chain_panel(chain, names)
```

Figure 1 shows the full parameter chains for all 10,000 MCMC simulations. The algorithm takes a few simulations to reach the correct distribution, which is clearly seen by the jump at the beginning. This is why we typically remove the first part of the chain to allow for burn-in. We make another plot, except this time we have removed the first part of the chain.

```
mcpl.plot_chain_panel(chain[burnin:,:], names)
```

Figure 2 shows the burned-in parameter chains based on the final 5,000 MCMC simulations. We observe that the distribution of parameter values appears to be consistent for the entire range of sampling shown, which supports the conclusion that we have converged to the posterior distribution. To visualize the distribution, we use the `plot_density_panel` method.

```
mcpl.plot_density_panel(chain[burnin:,:], names)
```

Figure 3 shows the marginal posterior parameter densities. The densities are generated using a Kernel Density Estimation (KDE) algorithm based on the parameter chains shown in Figure 2. The distributions appear to be nominally Gaussian in nature; however, that is not a requirement when running MCMC. One more chain diagnostic that we commonly consider is with regard to parameter correlation. We visualize the parameter correlation using the `plot_pairwise_correlation_panel` method.

```
mcpl.plot_pairwise_correlation_panel(
      chain[burnin:, :], names)
```

Figure 4 shows the pairwise parameter correlation based on the sample history of the MCMC simulation. Essentially, we take the points from the chain seen in Figure 2 and plot the matching points for m and b against one another. As seen in Figure 4, there appears to be a negative correlation between the two parameters; however, it is not particularly strong. The MCMC approach has no issues with correlated parameters, so these results are fine. Where you have to be careful is when the pairwise correlation shows a nearly single-valued relationship of some kind. By single-valued, we mean that the value of one parameter can be used to directly determine the other, e.g., if the pairwise correlation revealed a completely straight line.

Now that we have distributions for the parameters, we want to know how that uncertainty propagates through the model. Within pymcmcstat, the user has the ability to generate credible and prediction intervals. Credible intervals represent the distribution of the model output based simply on propagating the uncertainty from the parameter distributions. In contrast, prediction intervals also include uncertainty that arises due to observation errors ε_i. The following example code can be used to generate and plot credible and prediction intervals using pymcmcstat

```
def modelfun(pdata, theta):
    m, b = theta
```

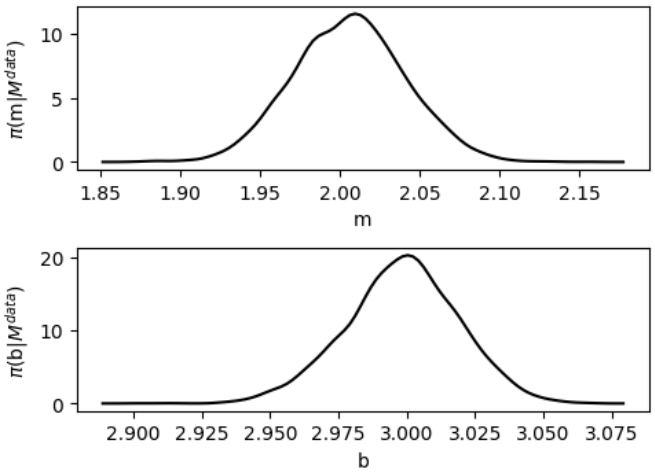


Fig. 3: *Marginal posterior parameter densities for linear model.*

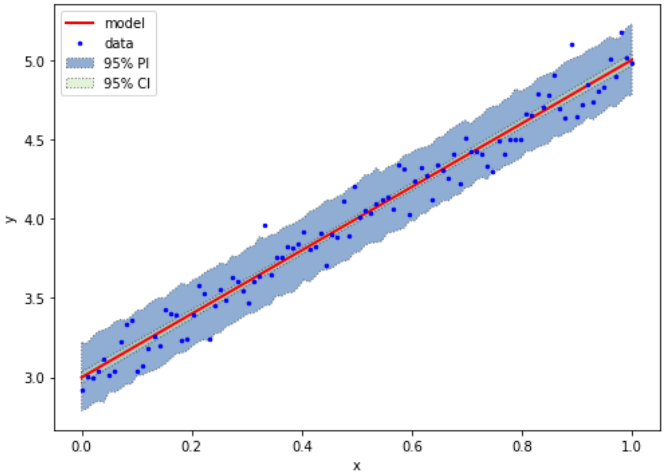


Fig. 5: *95% credible and prediction intervals for linear model.*

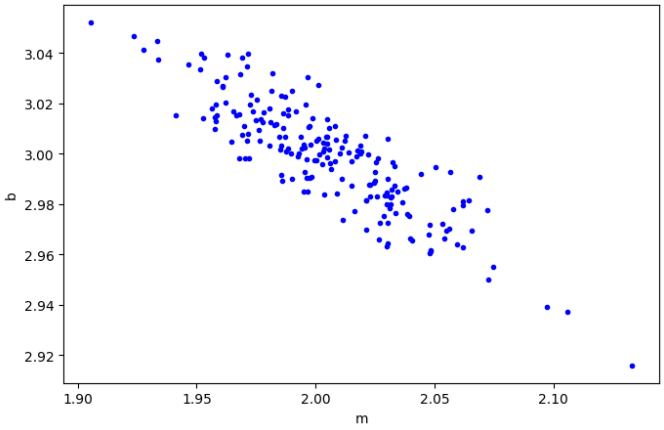


Fig. 4: *Pairwise correlation between sampling points for linear model.*

```
    x = pdata.xdata[0]
    y = m*x + b
    return y

mcstat.PI.setup_prediction_interval_calculation(
    results=results,
    data=mcstat.data,
    modelfunction=modelfun,
    burnin=burnin)
mcstat.PI.generate_prediction_intervals(
  calc_pred_int=True)
# plot prediction intervals
fg, ax = mcstat.PI.plot_prediction_intervals(
            adddata=True,
            plot_pred_int=True)
ax[0].set_ylabel('y')
ax[0].set_xlabel('x')
```

The procedure takes a subsample of the MCMC chain, evaluates the model for each sampled parameter set, and sorts the output to generate a distribution.

Figure 5 shows the 95% credible and prediction intervals. We observe that the credible intervals are fairly narrow, which is not surprising given the small amount of uncertainty in the parameter values (standard deviations of 0.03 and 0.02 for m and b, respectively). This is not always the case, especially in instances where there is unknown or missing physics in the model. However, we generated fictitious data using the model, so these results are reasonable. Prediction intervals quantify the probability of observing future numerical predictions or experimental observations because they include both parameter and observation uncertainty. For a 95% prediction interval, we expect a future observation to fall within that region 95% of the time. As a general check, we note that approximately 95% of the data appears to be inside the prediction interval shown in Figure 5, which is consistent with what we expect.

This concludes the basic example and highlights the workflow of how pymcmcstat could be used for a scientific problem. Note, this example highlighted a linear model; however, the algorithm is also applicable to nonlinear models, examples of which are discussed in subsequent sections.

Metropolis Algorithms

For those unfamiliar with Metropolis algorithms, we have provided a brief overview of the procedure. For each step of the MCMC simulation, a new set of parameter values are proposed q^*. We accept or reject q^* based on comparison with results obtained with the previous parameter set q^{k-1}. To do this we calculate the acceptance ratio

$$\alpha = \frac{\mathcal{L}(F^{obs}(i)|q^*)\pi_0(q^*)}{\mathcal{L}(F^{obs}(i)|q^{k-1})\pi_0(q^{k-1})}. \tag{3}$$

We observe that (3) compares the unscaled posterior probabilities. Essentially, we are computing whether q^* or q^{k-1} is more likely. For uniform prior distributions, this simplifies to comparing the likelihood function. For the Gaussian likelihood function (2), a smaller sum-of-squares error implies a larger likelihood. So, if the error is reduced by evaluating the model with q^*, the acceptance ratio will have a value $\alpha > 1$. In that case we accept the parameters and set $q^k = q^*$. In contrast, if the error increases (i.e., the likelihood decreases), the acceptance ratio becomes $\alpha < 1$. Rather than outright reject parameter sets that increase error, we conditionally accept q^* if $\alpha > U(0,1)$ (random value from a uniform distribution between 0 and 1). In this way we will often accept values that yield similar errors because the acceptance ratio will be closer to 1. Otherwise, we define the next simulation parameter set to be equal to the previous; i.e., $q^k = q^{k-1}$.

Candidates, q^*, are generated by sampling from a proposal distribution, which accounts for parameter correlation. In an ideal

TABLE 1: *Metropolis algorithms available in pymcmcstat.*

Algorithm	
MH	Metropolis-Hastings
AM	Adaptive Metropolis
DR	Delayed Rejection
DRAM	DR + AM

case one can adapt the proposal distribution as information is learned about the posterior distribution from accepted candidates. This is referred to as adaptive Metropolis (AM) and it is implemented in pymcmcstat using the algorithm presented in [HST+01]. Another desirable feature in Metropolis algorithms is to include delayed rejection (DR), which helps to stimulate mixing within the sampling chain. Good mixing simply means that the simulation is switching between points frequently and not stagnating on a single value; i.e., $q^k = q^{k-1}$ for many simulations in a row. This has been implemented using the algorithm presented in [HLMS06]. A summary of the Metropolis algorithms available inside pymcmcstat is presented in Table 1.

Options and Settings

Below we provide a brief summary of common features and explanations of how a user might implement them for a particular problem. As shown in the basic example, the user must define the options before running the simulation. The following code segment shows several additional simulation features that a user might find useful.

```
mcstat.simulation_options.define_simulation_options(
    nsimu=10.0e3,  # No. of MCMC simulations
    method='dram',  # Metropolis algorithm
    updatesigma=True,  # Update obs. error var.
    savedir='mcmc_chains',  # Output dir.
    save_to_bin=True,  # Save chains to binary
    save_to_txt=True,  # Save chains to text
    savesize=int(1.0e3),  # Saving intervals
    waitbar=False,  # Display progress bar
    verbosity=0,  # Level of display while running
    )
```

The list of available Metropolis algorithms is found in Table 1, and the user can change it via the `method` keyword argument. To update the observation error variance, σ^2, one sets `updatesigma=True`. The ability to update σ^2 is a direct result of the form of the likelihood function, and the reader is referred to [Smi14] for more details.

Several arguments relate to the ability to save results into a running log file. As the simulation runs, it periodically appends the sampling chain to a file. In this case, it will create binary (`save_to_bin=True`) and text (`save_to_txt=True`) files in a directory (`savedir='mcmc_results'`) and append the latest set of chain values every 1,000 simulations (`savesize=int(1.0e3)`). This can be extremely useful when running simulations over a long period of time. The user can run diagnostics on the latest set of chain results while the simulation is still running. For more details regarding this feature please see the tutorial on using Chain Log Files[3].

A progress bar will be displayed while the simulation runs; however, it is easily turned off by setting `waitbar=False`. Similarly, the program displays certain features depending on the level of `verbosity` specified. Setting `verbosity=0` suppresses all text output display. More information will be presented as you increase the value of `verbosity`.

Additional options are available for specifying the initial parameter covariance matrix (proposal distribution), adaptation interval, stages of delayed rejection, as well as outputting results to a JSON file. For more details regarding the options available in pymcmcstat, the reader is referred to the pymcmcstat documentation[4] and tutorials[5]. Next, we will outline some specific scientific problems in which pymcmcstat has been utilized to gain insight regarding model limitations in light of uncertainty.

Case Studies

Viscoelastic Modeling of Dielectric Elastomers

Dielectric elastomers are a type of smart material commonly implemented within an adaptive structure, which provide unique capabilities for control of a structure's shape, stiffness, and damping [Smi05]. These capabilities make them suitable for a wide variety of applications, including robotics, flow control, and energy harvesting [LG01], [CIS11]. Accurately modeling this material presents many challenges in light of its viscoelastic behavior. Viscoelastic materials exhibit a time-dependent strain response, which can vary significantly with the rate at which the material is being deformed [RC+03]. To help visualize this behavior, Figure 6 shows uni-axial experimental data for the elastomer Very High Bond (VHB) 4910. This highlights how as the material is deformed (i.e., stretch) you see a different stress response depending on the rate of deformation (i.e., stretch rate). Furthermore, at each rate you see two lines. The upper line reflects the material stress response as it is being loaded and the lower line is the stress as it is being relaxed. The gap between loading and relaxing is called hysteresis and is commonly seen in viscoelastic materials like this. For more details regarding the experimental procedure used to generate this data, the reader is referred to [MHSO15].

A variety of models can be used when modeling the behavior of these materials, but the details are beyond the scope of this paper. We implement a model of the form $F(i;q)$ to predict the nominal stress response during the loading and unloading of the material. The model depends on the parameter set

$$q = [G_c, G_e, \lambda_{max}, \eta, \gamma], \tag{4}$$

where each parameter helps describe a certain aspect of the physics that we are interested in modeling. Details regarding these models can be found in [DG13] and [MHSO15]. We calibrate the model with respect to the experimental data collected at $\dot{\lambda} = 0.67$ Hz as shown in Figure 6.

We can perform the MCMC simulation using the basic procedure previously outlined. For this particular case study, we wish to point out several specific devices that were used, and a full implementation of the code for this problem can be found in the Viscoelasticity Tutorial[6]. To begin, we point out the potential advantages of using pymcmcstat in conjunction with models written in faster computing languages.

In any sampling based method, computational efficiency is extremely important, and most of your computational time will

3. https://nbviewer.jupyter.org/github/prmiles/pymcmcstat/blob/master/tutorials/saving_to_log_files/Chain_Log_Files.ipynb

4. https://pymcmcstat.readthedocs.io/

5. https://nbviewer.jupyter.org/github/prmiles/pymcmcstat/blob/master/tutorials/index.ipynb

6. https://nbviewer.jupyter.org/github/prmiles/pymcmcstat/blob/master/tutorials/viscoelasticity/viscoelastic_analysis_using_ctypes.ipynb

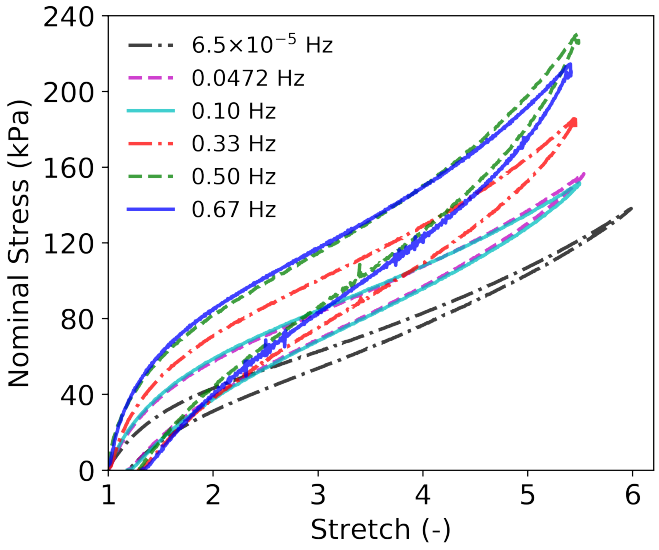


Fig. 6: *Experimental data for VHB 4910. The frequencies refer to different rates of deformation, or in this case different stretch rates, $\dot{\lambda}$.*

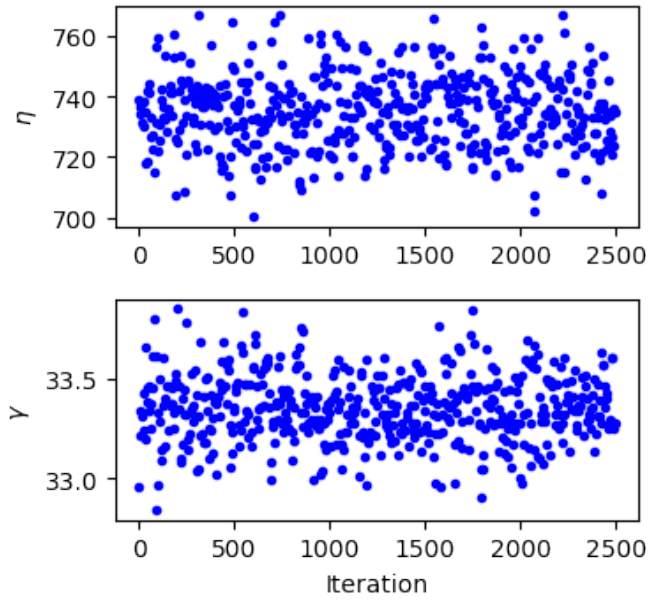


Fig. 7: *Parameter chains obtained with 2.5×10^3 realizations of the elastomer model.*

be spent in evaluating the model. We note that computational performance can be significantly improved by writing the model functions in C++ or Fortran. You can easily call these functions by utilizing the ctypes package, and an example of how to do this with pymcmcstat can be found in the Viscoelasticity Tutorial. For example, the elastomer model implemented here was written in both Python and C++. The average run time for a single model evaluation using C++ was approximately 0.09 ms whereas the Python implementation took over 8 ms. This particular model is reasonably fast in both languages, but we wished to point out the advantage of using more efficient code for the model evaluation.

Another item that commonly arises in model calibration is that not all your parameters are identifiable. Determination of identifiable parameters is typically done using some type of sensitivity analysis, which is beyond the scope of this paper. For this example, let us suppose that the first three parameters in q have known, fixed values and therefore should not be included in the sampling chain of the MCMC simulation. As they are fixed values, one could simply hard code the parameters into the sum-of-squares function like this

```
def ssfun(q, data):
    # Assign model parameters
    Gc, Ge, lam_max = 7.55, 17.7, 4.83
    eta, gamma = q
    # evaluate elastomer model
    ...
```

This solution is not ideal as you may later decide to include those parameters as part of the calibration. To accommodate models with fixed parameters, pymcmcstat allows the user to specify whether or not to include parameters in the sampling process. This is accomplished by specifying `sample=False` as follows

```
# define model parameters
mcstat.parameters.add_model_parameter(
    name='$G_c$',
    theta0=7.55,
    sample=False)
mcstat.parameters.add_model_parameter(
    name='$G_e$',
    theta0=17.7,
    sample=False)
mcstat.parameters.add_model_parameter(
    name='$\\lambda_{max}$',
    theta0=4.83,
    sample=False)
mcstat.parameters.add_model_parameter(
    name='$\\eta$',
    theta0=708)
mcstat.parameters.add_model_parameter(
    name='$\\gamma$',
    theta0=31)
```

This now allows the user to define their sum-of-squares function without hard coded values for the first three parameters.

```
def ssfun(q, data):
    # Assign model parameters
    Gc, Ge, lam_max, eta, gamma = q
    # evaluate elastomer model
    ...
```

The final item for this case study relates to assessing chain convergence. Previously, we outlined a variety of plotting methods available for looking at the sampling history and parameter correlation. We also mentioned various statistical measures, such Geweke's convergence diagnostic and autocorrelation time. The chain panel shown in Figure 7 appears to be converged, but there is a possibility that the algorithm is stuck in a local minimum. If you run the simulation longer, then you may see a jump in the chain as it finds another local minimum. For a more rigorous assessment of chain convergence, the user can generate multiple sets of chains and use Gelman-Rubin diagnostics [GR+92]. An example of how to generate multiple chains with pymcmcstat can be found in the Running Parallel Chains Tutorial[7], which also includes information on how to calculate Gelman-Rubin diagnostics.

7. https://nbviewer.jupyter.org/github/prmiles/pymcmcstat/blob/master/tutorials/running_parallel_chains/running_parallel_chains.ipynb

Fig. 8: *Simulated 250m × 178m block of downtown Washington D.C.*

Radiation Source Localization

Efficient and accurate localization of special nuclear material (SNM) in urban environments is a vitally important task to national security and presents many unique computational challenges. A realistic problem requires accounting for radiation transport in 3D, using representative nuclear cross-sections for solid materials, and simulating the expected interaction with a network of detectors. This is a non-trivial task that highlights the importance of surrogate modeling when high-fidelity models become computationally intractable for sampling based methods. For the purpose of this example, we will highlight some previous research that utilizes a ray-tracing approach in 2D. We simulate a 250m × 178m block of downtown Washington D. C. as shown in Figure 8.

We implement a highly simplified radiation transport model which ignores scattering. The model accounts for signal attenuation that is caused by distance as well as interference from buildings that are in the path between the source and detector location. This ray tracing model is implemented in the Python package gefry3[8]. Additional details regarding this research can be found in [HM19].

As with the viscoelasticity case study, we only highlight several key features for solving this problem with pymcmcstat. The complete code can be found in the Radiation Source Localization Tutorial[9]. The first item we wish to highlight is the ability to pass additional information into the sum-of-squares function by utilizing the `user_defined_object` feature of the data structure.

```
# setup data structure for dram
mcstat.data.add_data_set(
    x=np.zeros(observations.shape),
    y=observations,
    user_defined_object=[
        model,
        background,
    ],
)
```

In this case, we have created an object which is a list with two elements: 1) the radiation transport model and 2) the background radiation. These items are easily accessed within the sum-of-squares function.

```
# Radiation Sum of Squares Function
def radiation_ssfun(theta, data):
    x, y, I = theta
    model, background = data.user_defined_object[0]
    output = model((x, y), I) + background
    res = data.ydata[0] - output
    ss = (res ** 2).sum(axis = 0)
    return ss
```

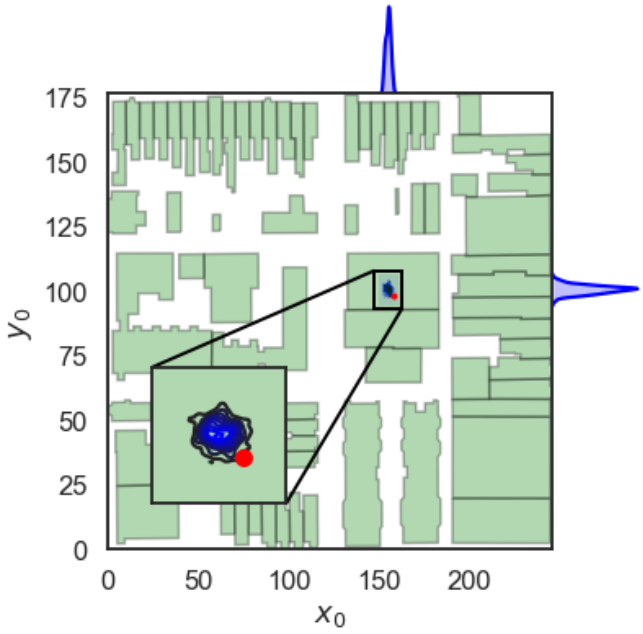


Fig. 9: *Marginal posteriors from MCMC simulation presented in urban environment. Actual source location is denoted by the red circle.*

A Bayesian approach to source localization provides us with several very practical results. Firstly, there are multiple regions of the domain that will yield comparable detector measurements, so assigning probabilities to various locations is more realistic than a single point estimate. If one can infer regions of higher probability, it can then motivate the placement of new detectors in the domain or possibly allow for a team with handheld detectors to complete the localization process. Given the challenges of modeling the radiation transport physics, it is extremely useful to visualize the potential source locations in light of the underlying uncertainty. Figure 9 shows the marginal posterior densities, where it is clearly seen that the posteriors are very close to the true source location. We note that this plot was generated using the mcmcplot package [Mil19a], and the required code can be found in the previously referenced Radiation Source Localization Tutorial.

This is a very simplified case, but it highlights another unique problem in which pymcmcstat can be used to gain insight regarding uncertainty.

Concluding Remarks

The pymcmcstat package presents a robust platform from which to perform a wide array of Bayesian inverse problems using the Delayed Rejection Adaptive Metropolis (DRAM) algorithm. In this paper we have provided a basic description of Markov Chain Monte Carlo (MCMC) methods and outlined a general example

8. https://github.com/jasonmhite/gefry3

9. https://nbviewer.jupyter.org/github/prmiles/pymcmcstat/blob/master/tutorials/radiation_source_localization/radiation_source_localization.ipynb

of how to implement pymcmcstat. Furthermore, we highlighted aspects of two distinct areas of scientific study where MCMC methods provided enhanced understanding of the underlying physics.

To improve the overall usefulness of the pymcmcstat package, we will expand its functionality to allow for user-defined likelihood and prior functions (currently limited to Gaussian). We designed the package to serve as a Python alternative for the MATLAB toolbox mcmcstat, so it is important to maintain the features of the original user interface for ease of transition from one platform to another. Overall, the package is applicable to a wide variety of scientific problems, and provides a nice interface for users who are potentially new to Bayesian methods.

Acknowledgments

This research was supported by the Department of Energy National Nuclear Security Administration (NNSA) under the Award Number DE-NA0002576 through the Consortium for Nonproliferation Enabling Capabilities (CNEC). Additional support was provided by the Air Force Office of Scientific Research (AFOSR) through Award Number FA9550-15-1-0299.

REFERENCES

[Box76] George EP Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[BR98] Stephen P Brooks and Gareth O Roberts. Assessing convergence of markov chain monte carlo algorithms. *Statistics and Computing*, 8(4):319–335, 1998.

[CIS11] Louis N Cattafesta III and Mark Sheplak. Actuators for active flow control. *Annual Review of Fluid Mechanics*, 43:247–272, 2011. doi:10.1146/annurev-fluid-122109-160634.

[DG13] Jacob D Davidson and NC Goulbourne. A nonaffine network model for elastomers undergoing finite deformations. *Journal of the Mechanics and Physics of Solids*, 61(8):1784–1797, 2013. doi:10.1016/j.jmps.2013.03.009.

[GR+92] Andrew Gelman, Donald B Rubin, et al. Inference from iterative simulation using multiple sequences. *Statistical science*, 7(4):457–472, 1992. doi:10.1214/ss/1177011136.

[HLMS06] Heikki Haario, Marko Laine, Antonietta Mira, and Eero Saksman. DRAM: Efficient adaptive mcmc. *Statistics and computing*, 16(4):339–354, 2006. doi:10.1007/s11222-006-9438-0.

[HM19] Jason Hite and John Mattingly. Bayesian metropolis methods for source localization in an urban environment. *Radiation Physics and Chemistry*, 155:271–274, 2019. doi:10.1016/j.radphyschem.2018.06.024.

[HST+01] Heikki Haario, Eero Saksman, Johanna Tamminen, et al. An adaptive metropolis algorithm. *Bernoulli*, 7(2):223–242, 2001.

[LG01] Malcolm E Lines and Alastair M Glass. *Principles and applications of ferroelectrics and related materials*. Oxford university press, 2001.

[MHSO15] Paul R. Miles, Michael Hays, Ralph C. Smith, and William S. Oates. Bayesian uncertainty analysis of finite deformation viscoelasticity. *Mechanics of Materials*, 91:35–49, 2015. doi:10.1016/j.mechmat.2015.07.002.

[Mil19a] Paul Miles. mcmcplot, April 2019. doi:10.5281/zenodo.2638340.

[Mil19b] Paul R. Miles. A python package for bayesian inference using delayed rejection adaptive metropolis. *Journal of Open Source Software*, 4(38):1417, 2019. doi:10.21105/joss.01417.

[MRR+53] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[RC+03] Michael Rubinstein, Ralph H. Colby, et al. *Polymer physics*, volume 23. Oxford university press New York, 2003.

[Smi05] Ralph C. Smith. *Smart material systems: model development*. SIAM, 2005.

[Smi14] Ralph C. Smith. *Uncertainty Quantification: Theory, Implementation, and Applications*, volume 12. SIAM, 2014.

PyLZJD: An Easy to Use Tool for Machine Learning

Edward Raff[§‡*], Joe Aurelio[‡§], Charles Nicholas[‡]

Abstract—As Machine Learning (ML) becomes more widely known and popular, so too does the desire for new users from other backgrounds to apply ML techniques to their own domains. A difficult prerequisite that often confounds new users is the feature creation and engineering process. This is especially true when users attempt to apply ML to domains that have not historically received attention from the ML community (e.g., outside of text, images, and audio). The Lempel Ziv Jaccard Distance (LZJD) is a compression based technique that can be used for many machine learning tasks. Because of its compression background, users do not need to specify any feature extraction, making it easy to apply to new domains. We introduce PyLZJD, a library that implements LZJD in a manner meant to be easy to use and apply for novice practitioners. We will discuss the intuition and high-level mechanics behind LZJD, followed by examples of how to use it on problems of disparate data types.

Index Terms—compression, complex data, machine learning

Introduction

Machine Learning (ML) has become an increasingly popular tool, with libraries like Scikit-Learn [PVG+11] and others [CG16], [Raf17], [MBY+16], [HFH+09] making ML algorithms available to a wide audience of potential users. However, ML can be daunting for new and amateur users to pick up and use. Before even considering what algorithm should be used for a given problem, feature creation and engineering is a prerequisite step that is not easy to perform, nor is it easy to automate.

In normal use, we as ML practitioners would describe our data as a matrix $\boldsymbol{X}$ that has n rows and d columns. Each of the n rows corresponds to one of our data points (i.e., an example), and each of the d columns corresponds to one of our features. Using cars as an example, we may want to know what color a car is, how old it is, or its odometer mileage, as features. We want to have these features in every row n of our matrix so that we have the information for every car. Once done, we might train a model $m(\cdot)$ to perform a classification problem (e.g., is the car an SUV or sedan?), or use some distance measure $d(\cdot,\cdot)$ to help us find similar or related examples (e.g., which used car that has been sold is most like my own?).

The question becomes, how do we determine what to use as our features? One could begin enumerating every property a car might have, but that would be time consuming, and not all of the features would be relevant to all tasks. If we had an image of a car, we might use a Neural Network to help us extract information or find similar looking images. But if one does not have prior experience with machine learning, these tasks can be daunting. For some types of complex data, feature engineering can be challenging even for experts.

To help new users avoid this difficult task, we have developed the PyLZJD library. PyLZJD makes it easy to get started with ML algorithms and retrieval tasks without needing any kind of feature specification, selection, or engineering from the user. Instead, a user represents their data as a file (i.e., one file for every data point, for n total files). PyLZJD will automatically process the file and can be used with Scikit-Learn to tackle many common tasks. While PyLZJD will likely not be the best method to use for most problems, it provides an avenue for new users to begin using machine learning with minimal effort and time.

The Lempel Ziv Jaccard Distance

LZJD stands for "Lempel Ziv Jaccard Distance" [RN17a] and is the algorithm implemented in PyLZJD. LZJD takes a byte or character sequence x (i.e., a "string"), converts it to a set of substrings, and then converts the set into a *digest*. This digest is a fixed-length summary of the input sequence, which requires a total of k integers to represent. We can then measure the similarity of digests using a distance function, and we can trade accuracy for speed and compactness by decreasing k. We can optionally convert this digest into a vector in Euclidean space, allowing greater flexibility to use LZJD with other machine learning algorithms.

The inspiration and high-level understanding of LZJD comes from compression algorithms. Let $C(\cdot)$ represent your favorite compression algorithm (e.g., zip or bz2), which takes an input x and produces a compressed version $C(x)$. Using a decompressor, you can recover the original object or file x from $C(x)$. The purpose of this compression is to reduce the size of the file stored on disk. So if $|x|$ represents how many bytes it takes to represent the file x, the goal is that $|C(x)| < |x|$.

What if we wanted to compare the similarity of two files, x and y? We can use compression to help us do that. Consider two files x and y, with absolutely no shared content. Then we would expect that if we concatenated x and y together to make one larger file, $x\|y$, then compressing the concatenated version of the files should be about the same size as the files compressed separately, $|C(x\|y)| = |C(x)| + |C(y)|$. But what if $|C(x\|y)| < |C(x)| + |C(y)|$? For that to be true, there must be some overlapping content between x and y that our compressor $C(\cdot)$ was able to reuse in order to achieve a smaller output. The more similarity between x and y, the greater difference in file size we should see. In which case, we could use the ratio of compressed file lengths to tell us how similar

* Corresponding author: raff_edward@bah.com
§ Booz Allen Hamilton
‡ University of Maryland, Baltimore County

the files are. We could call this a "Compression Distance Metric" [KLR04] as shown in Equation 1, where CDM(x,y) returns a smaller value the more similar x and y are, and a larger value if they are different.

$$\text{CDM}(x,y) = \frac{C(x\|y)}{|C(x)|+|C(y)|} \tag{1}$$

The CDM distance we just described gives the intuition behind LZJD. That we can use compression algorithms to measure the similarity between arbitrary files. CDM has been used to perform time series clustering and classification [KLR04]. A large number of compression based distance measures have been proposed [SB06] and used for tasks such as DNA clustering [LCL+04], image retrieval [Tra07], and malware classification [Bor15].

Mechanics of LZJD

While the above strategy has seen much success, it also suffers from drawbacks. Using a compression algorithm for every similarity comparison makes prior methods slow, and the mechanics of standard compression algorithms are not optimized for machine learning tasks. Equation 1 also does not have the properties of a true distance metric[1], which can lead to confusing behavior and prevents using tools that rely on these properties. LZJD rectifies these issues by converting a specific compression algorithm, LZMA, into a dedicated distance metric [RN17a]. LZJD is fast enough to use for larger datasets and maintains the properties of a true distance metric. LZJD works by first creating the compression dictionary of the Lempel Ziv algorithm [LZ76].

```
def lzset(b): #code for string case only
    s = set()
    start = 0
    end = 1
    while end <= len(b):
        b_s = b[start:end]
        if b_s not in s:
            s.add(b_s)
            start = end
        end += 1
    return s

def sim(A, B): # A & B should be set objects
    return len(A & B) / len(A | B)
```

The `lzset` method shows the Lempel compression dictionary creation process. Since LZJD cares about similarity as a direct goal, we do not put in the extra work or code normally required to make an effective compressor. Instead, we simply create a Python set of many different sub-strings of the input sequence `b`. Because the `lzset` method gives us a set of objects, we use the well-known Jaccard similarity to measure how close the two sets are. This is defined in the `sim` method above, and mathematically in Equation 2.

$$J(A,B) = \frac{|A\cap B|}{|A\cup B|} = \frac{|A\cap B|}{|A|+|B|-|A\cap B|} \tag{2}$$

The distance $d(A,B) = 1-J(A,B)$ is a valid metric, and thus provides all the tools necessary to measure the similarity between arbitrary sequences or files. If a and b represent different sequences, their LZJD is computed as:

```
dist = 1.0-sim(lzset(a),lzset(b))
```

1. The properties of a true distance metric are symmetry, indiscernibility, and the triangle inequality.

While the procedure above will implement the LZJD algorithm, it does not include the speedups that have been incorporated into PyLZJD. Following [RN17a] we use Min-Hashing [BCFM98] to convert a set A into a more compact representation A', which is of a fixed size k (i.e., $|A'|=k$) but guarantees that $J(A,B)\approx J(A',B')$[2]. [RN18] reduced computational time and memory use further by mapping every sub-sequence to a hash and performing `lzset` construction using a rolling hash function to ensure every byte of input was only processed once. To handle class imbalance scenarios, a stochastic variant of LZJD allows over-sampling to improve accuracy [RN17b]. All of these optimizations were implemented with Cython [BBC+11] in order to make PyLZJD as fast as possible.

Vectorizing Inputs

The LZJD algorithm as discussed so far provides only a distance metric. This is valuable for search and information retrieval problems, many clustering algorithms, and k-nearest-neighbor style classification, but it does not avail ourselves to all the algorithms that would be available in Scikit-Learn. Prior work proposed one method of vectorizing LZSets [RN17b] based on feature hashing [WDL+09], where every item in the set is mapped to a random position in a large and high dimensional input (they used $d=2^{20}$). For new users, we want to avoid such high dimensional spaces to avoid the *curse of dimensionality* [Bel57], a phenomena that makes obtaining meaningful results in higher dimensions difficult.

Working in such high dimensional spaces often requires greater consideration and expertise. To make PyLZJD easier for novices to use, we have developed a different vectorization strategy. To make this possible, we use a new version of Min-Hashing called "SuperMinHash", [Ert17]. The new SuperMinHash is up to 40% slower compared to the prior method, but enables us to use what is known as b-bit minwise hashing to convert sets to a more compact vectorized representation [LK11]. Since $k \leq 1024$ in most cases, and $b \leq 8$, we arrive at a more modest $d = k \cdot b \leq 8,192$. By keeping the dimension smaller, we make PyLZJD easier to use and a wider selection of algorithms from Scikit-Learn should produce reasonable results.

Over-Sampling Data

Another feature introduced in [RN17b] is the ability to stochastically over-sample data to create artificially larger datasets. This is particularly useful when working with imbalanced datasets. Given a value `false_seen_prob`, their approach modifies the inner if statement of `lzset` to falsely "see" a sub-string that it has not seen before. This is a one line change that looks like the following:

```
if b_s not in s
  and random.uniform() > false_seen_prob:
```

By doing so, the set of sub-strings returned is altered. However, the altered set is still true to the data in that every string in the set is a real and valid sub-string from the corpus. This works because the Lempel Ziv dictionary creation is sensitive to small changes in the input, so a few small alterations can propagate forward and cause a number of differences in the entire process. By making the condition random, we can repeat the process several times and get different results each time. This provides additional example diversity that can help train a model. When `false_seen_prob`

2. The bottom-k approach is used by default, where one hash $h(\cdot)$ is applied to every item in the set, and the bottom-k values according to $h(\cdot)$ are selected.

= 0, we get the standard LZJD output. To perform oversampling, we recommend using small values like `false_seen_prob` $\leq$ 0.05.

Using PyLZJD

Now that we have given the intuition and described how LZJD works, we show three examples of how PyLZJD performs machine learning, without having to specify a feature processing pipeline. PyLZJD, along with complete versions of these examples, can be found at https://github.com/EdwardRaff/pyLZJD.

To use PyLZJD, at most three functions need to be imported, as shown below.

```
from pyLZJD import digest, sim, vectorize
```

These three functions work as follows:

- `digest(b, hash_size=1024, mode=None, processes=-1, false_seen_prob=0.0)`: takes in (1) a string as data to convert to a digest or (2) a path to a file and converts the file's content to an LZJD digest. If a list is given as input, each element of the list will be processed to return a list of digests.[3]
- `vectorize(b, hash_size=1024, k=8, processes=-1, false_seen_prob=0.0)`: works the same as digest, but instead of returning a list, returns a numpy array representing a feature vector.
- `sim(A, B)`: takes two LZJD digests, and returns the similarity score between two files. 1.0 indicating they are exactly similar, and 0.0 indicating no similarity.

The above is all that is needed for practitioners to use PyLZJD in their code. Below we will go through three examples of how to use these functions in conjunction with Scikit-Learn to get decent results on these problems. For new users, we recommend considering LZJD as a first-pass easy-to-use algorithm so long as the length of the input data is 200 bytes/characters or more. This recommendation comes from the fact that LZJD is compression based, and it is difficult to compress very short sequences. A quick test of LZJD's appropriateness, is to manually compress your data points (as files) with your favorite compression algorithm. If the files compress well, LZJD may work. If the files do not compress well, LZJD is less likely to work.

T5 Corpus Example

The first example we use is a dataset called T5, which has historically been used for computer forensics [Rou11]. It contains 4,457 files that are of one of 8 different file types: html, pdf, text, doc, ppt, jpg, xls, or gif. As a simple first step to using PyLZJD, we will attempt to classify a file as one of these 8 file types. Our code starts by collecting the paths to each file into a list `X_paths`. Creating a LZJD digest for each of these files is simple; call the `digest` function as shown below:

```
X_hashes = digest(X_paths, processes=-1)
```

The processes argument is optional. By setting it to -1, as many processor cores as are available are used. If set to any positive value n, then n cores will be used. A list of digests will be returned with the same corresponding order as the input. The `digest` function will automatically load every file path from disk, and perform the LZJD process outlined above.

For this first example, we will stick to using LZJD as a similarity tool and distance metric. When you want to use distance based algorithms, you want to use the `digest` and `sim` functions instead of `vectorize`. `vectorize` is less accurate and slower when computing distances.

To use LZJD's digest with Scikit-Learn, we need to massage the files into a form that it expects. Scikit-Learn needs a distance function between data stored as a list of vectors (i.e., a matrix X). However, our digests are not vectors in the way that Scikit-Learn understands them, so Scikit-Learn needs to be told how to properly measure distances when using LZJD. An easy way to do this[4], which is compatible with other specialized distance a user may want to leverage, is to create a 1-D list of vectors. Each vector will store the index of its digest in the created `X_hashes` list. Then we create a distance function which uses the index and returns the correct value. While wordy to explain, it takes only a few lines of code:

```
#This will be the vetor given to Scikit-Learn
X = [ [i] for i in range(len(X_hashes))]

#sklearn will give us two vectors a and b from 'X'
def lzjd_dist(a, b):
    #Each has len(a) = 1, so only one value to grab
    #The stored value tells us which index
    #has 'our' digest
    digest_a = X_hashes[int(a[0])]
    digest_b = X_hashes[int(b[0])]
    #Now that we have the digests, compute a
    #distance measure.
    return 1.0-sim(digest_a, digest_b)
```

This is all we need to use the tools built into Scikit-learn. For example, we can perform k-nearest-neighbor classification with cross-validation to see how accurately we predict a file's type.

```
knn_model = KNeighborsClassifier(n_neighbors=5,
    algorithm='brute', metric=lzjd_dist)

scores = cross_val_score(knn_model, X, Y)
print("Accuracy: %0.2f (+/- %0.2f)"
    % (scores.mean(), scores.std() * 2))
```

The above code returns a value of 91% accuracy, where a majority-vote baseline returns 25%. This was all done without us having to specify anything about the associated file formats, how to parse them, or any feature engineering work. We can also leverage other distance metric based tools that Scikit-Learn provides. For example, we can use the t-SNE [MH08] algorithm to create a 2D embedding of our data that we can visualize with matplotlib. Using Scikit-Learn, this is only one line of code:

```
X_embedded = TSNE(n_components=2, perplexity=5,
    metric=lzjd_dist).fit_transform(X)
```

The resulting plot is shown in Figure 1. We see that the groups are mostly clustered into separate regions, but that there is a significant collection of points that were difficult to organize with their respective groups. While a tutorial on effective t-SNE use is beyond our scope, LZJD allows us to leverage t-SNE for immediate visual feedback and exploration.

3. `mode` controls which version of min-hashing is used. `None` for the standard hash, or `"SuperHash"` to use the approach that is compatible with vectorization.

4. This approach is how the Scikit-learn developers recomend using other non-standard distance metrics. For example, the Scikit-learn FAQ shows how to use this approach for doing edit-distance over strings.

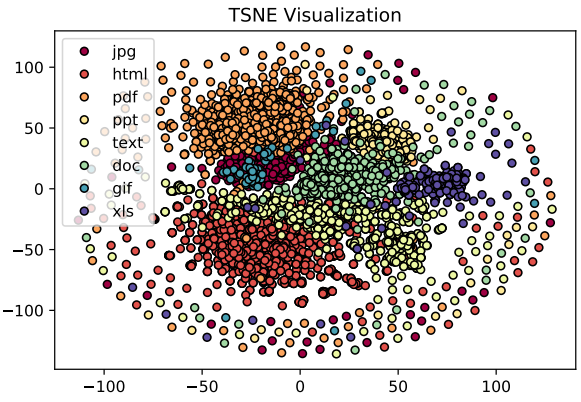


Fig. 1: *Example of t-SNE visualization created using LZJD. Best viewed digitally and in color.*

Spam Image Classification

The prior example used files of varying types, which is similar to the problem domain that LZJD was developed for. In this example, we change the type of data and how we approach the problem. Here, our goal is to predict if an email image attachment is a *spam* image (i.e., undesirable) or a *ham* image (i.e., desirable - or at least, more desirable than spam). This dataset was collected in 2007 [DGEB07], with 3298 spam and 2021 ham images.

Fig. 2: *Example of ham (left) and spam (right) images from the dataset's website.*

We use the `vectorize` function to create feature vectors for each data point. Using `vectorize` instead of `digest` allows us to build models that avoid the nearest neighbor search, which can be slow and cumbersome to deploy. The trade off is we spend more time during the training phase of the algorithm. Doing this with PyLZJD is simple, and the below code snippet handles the work of creating the labels, loading the files, and creating feature vectors, again, without us having to specify anything about the input.

```
spam_paths = glob.glob("personal_image_spam/*")
ham_paths = glob.glob("personal_image_ham/*")

all_paths = spam_paths + ham_paths
yBad = [1 for i in range(len(spam_paths))]
yGood = [0 for i in range(len(ham_paths))]
y = yBad + yGood
X = vectorize(all_paths)
```

Now that we have feature vectors, we can train a Logistic Regression model to predict if a new image is a spam or not. The code to train and evaluate it (by several metrics) is:

```
X_train, X_test, y_train, y_test =
  train_test_split(X, y, test_size=0.2,
    random_state=42)

lgs = LogisticRegression(class_weight='balanced')
lgs.fit(X_train, y_train) #training our model

predicted = lgs.predict(X_test)

fpr, tpr, _ = metrics.roc_curve(y_test,
  (lgs.predict_proba(X_test)[:, 1]))
auc = metrics.auc(fpr, tpr)
print("Accuracy: %f" %
  lgs.score(X_test, y_test))
print("Precision: %f" %
  metrics.precision_score(y_test, predicted))
print("Recall: %f" %
  metrics.recall_score(y_test, predicted))
print("F1-Score: %f" %
  metrics.f1_score(y_test, predicted))
print("AUC: %f" % auc)
```

This produces an accuracy of about 94.6%, and an AUC of 98.7%. In the above code snippet, we included the `class_weight` parameter to address class imbalance in the data. There are more examples of spam images, which can bias a model toward calling most inputs "spam" by default. Using a 'balanced' class weight re-weights the data as if there was an equal number of examples of each class. With PyLZJD, you can perform a special type of over-sampling to help further reduce this impact and improve accuracy. Here is a simple version of this ability:

```
paths_train, paths_test, y_train, y_test =
  train_test_split(all_paths, y,
    test_size=0.2, random_state=42)

X_train_clean = vectorize(paths_train)
X_train_aug = vectorize(paths_train*10,
  false_seen_prob=0.05)
X_test = vectorize(paths_test)
```

In this code, `X_train_clean` constructs the training data in the normal manner. Alternatively, `X_train_aug` has over-sampled both the spam and ham training data 10 times. Normally, this would create 10 copies of the same vectors and have no impact on the solution learned. But, we added the `false_seen_prob` flag, which alters how the `lzset` is constructed: this flag turns on the stochastic component and you get a different result every call. We get a variety of different (but realistic) examples for each datapoint. If we train a new logistic regression model on this data, we get improved results (Table 1).

TABLE 1: *Results on training a Logistic Regression model for spam image detection. Over-sampled scores show results when 'false_seen_prob' is used.*

Metric	Score	Over-sampled Score
Accuracy	0.946	0.957
Precision	0.950	0.954
Recall	0.966	0.979
F1-Score	0.958	0.966
AUC	0.987	0.992

LZJD won't always be effective for images, and convolutional neural networks (CNNs) are a better approach if you need the best possible accuracy. However, this example demonstrates that LZJD can still be useful, and has been used successfully to find slightly altered images [Fj]. This example also shows how to build a more deployable classifier with PyLZJD and tackle class-imbalance situations.

Text Classification

As our last example, we will use a text-classification problem. While other methods will work better, the purpose is to show that LZJD can be used in a wide array of potential applications. For this, we will use the well-known 20 Newsgroups dataset, which is available in Scikit-Learn. We use this dataset because LZJD works best with longer input sequences. For simplicity we will stick with distinguishing between the newsgroup categories of 'alt.atheism' and 'comp.graphics'. An example of an email from the later group is shown below.

> By '8 grey level images' you mean 8 items of 1bit images? It does work(!), but it doesn't work if you have more than 1bit in your screen and if the screen intensity is non-linear.
>
> With 2 bit per pixel; there could be 1*c_1 + 4*c_2 timing, this gives 16 levels, but they are linear if screen intensity is linear. With 1*c_1 + 2*c_2 it works, but we have to find the best compinations -- there's 10 levels, but 16 choises; best 10 must be chosen. Different compinations for the same level, varies a bit, but the levels keeps their order.
>
> Readers should verify what I wrote... :-)

When a string is not a valid path to a file, PyLZJD will processes the string itself to create a digest. This simplifies working with strings, and getting results is as easy as:

```
X_train = vectorize(newsgroups_train.data)
X_test = vectorize(newsgroups_test.data)

clf = LogisticRegression()
clf.fit(X_train, newsgroups_train.target)

pred = clf.predict(X_test)
metrics.f1_score(newsgroups_test.target,
    pred, average='macro')
```

With the above code, we get an F_1 score of 83%. Using Scikit-Learn's TfidfVectorizer achieves an F_1 of 89%. The point here is that with pyLZJD we can get decent results without having to think about what kind of vectorization is being performed, and any string encoded data can be feed directly into the `vectorize` or `digest` functions to get immediate results.

Conclusion

We have shown, by example, how to use PyLZJD on a number of different datasets composed of raw binary files, images, and regular ASCII text. In all cases, we did not have to do any feature engineering or extraction to use PyLZJD, making application simpler and easier. This shortcut is particularly useful when feature specification is hard, such as raw file types, but can also make it easier for people to get into applying Machine Learning.

References

[BBC+11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011. doi:10.1109/MCSE.2010.118.

[BCFM98] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 327–336, New York, NY, USA, 1998. ACM. URL: http://doi.acm.org/10.1145/276698.276781, doi:10.1145/276698.276781.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bor15] Rebecca Schuller Borbely. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques*, pages 1–8, 2015. doi:10.1007/s11416-015-0260-0.

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: Reliable Large-scale Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. arXiv:1603.02754.

[DGEB07] Mark Dredze, Reuven Gevaryahu, and Ari Elias-Bachrach. Learning fast classifiers for image spam. In *Proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2007.

[Ert17] Otmar Ertl. SuperMinHash – A New Minwise Hashing Algorithm for Jaccard Similarity Estimation. *arXiv*, 2017. arXiv:arXiv:1706.05698v1.

[Fj] João Felipe Pontes Faria-joao. Detecção de Imagens Similares: Aplicabilidade de Ferramentas Software Livre de Hash de Similaridade de Uso Geral. URL: https://www.ipog.edu.br/revista-especialize-online/edicao-16-2018-dez/deteccao-de-imagens-similares-aplicabilidade-de-ferramentas-software-livre-de-hash-de-similaridade-de-uso-geral/.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update Mark. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, nov 2009. doi:10.1145/1656274.1656278.

[KLR04] Eamonn Keogh, Stefano Lonardi, and Chotirat Ann Ratanamahatana. Towards Parameter-free Data Mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 206–215, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1014052.1014077, doi:10.1145/1014052.1014077.

[LCL+04] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M.B. Vitanyi. The Similarity Metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004. arXiv:0111054, doi:10.1109/TIT.2004.838101.

[LK11] Ping Li and Arnd Christian König. Theory and Applications of B-bit Minwise Hashing. *Commun. ACM*, 54(8):101–109, aug 2011. URL: http://doi.acm.org/10.1145/1978542.1978566, doi:10.1145/1978542.1978566.

[LZ76] A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, jan 1976. URL: http://ieeexplore.ieee.org/document/1055501/, arXiv:0009084, doi:10.1109/TIT.1976.1055501.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D B Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL: http://jmlr.org/papers/v17/15-237.html.

[MH08] Laurens Van Der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[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: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[Raf17] Edward Raff. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research*, 18(23):1–5, 2017. URL: http://jmlr.org/papers/v18/16-131.html.

[RN17a] Edward Raff and Charles Nicholas. An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pages 1007–1015, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3097983.3098111, doi:10.1145/3097983.3098111.

[RN17b] Edward Raff and Charles Nicholas. Malware Classification and Class Imbalance via Stochastic Hashed LZJD. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 111–120, New York, NY, USA,

2017. ACM. URL: http://doi.acm.org/10.1145/3128572.3140446, doi:10.1145/3128572.3140446.

[RN18] Edward Raff and Charles K. Nicholas. Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, feb 2018. URL: https://doi.org/10.1016/j.diin.2017.12.004, arXiv:1708.03346, doi:10.1016/j.diin.2017.12.004.

[Rou11] Vassil Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8:S34–S41, 2011. doi:10.1016/j.diin.2011.05.005.

[SB06] D Sculley and Carla E Brodley. Compression and Machine Learning: A New Perspective on Feature Space Vectors. In *Proceedings of the Data Compression Conference*, DCC '06, page 332, Washington, DC, USA, 2006. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DCC.2006.13, doi:10.1109/DCC.2006.13.

[Tra07] Nicholas Tran. The normalized compression distance and image distinguishability. In Bernice E. Rogowitz, Thrasyvoulos N. Pappas, and Scott J. Daly, editors, *Proc. SPIE 6492, Human Vision and Electronic Imaging XII*, volume 64921D, feb 2007. URL: http://dx.doi.org/10.1117/12.704334, doi:10.1117/12.704334.

[WDL+09] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1113–1120, New York, New York, USA, 2009. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1553374.1553516, doi:10.1145/1553374.1553516.

Parkinson's Classification and Feature Extraction from Diffusion Tensor Images

Rajeswari Sivakumar‡*, Shannon Quinn‡

Abstract—Parkinson's disease (PD) affects over 6.2 million people around the world. Despite its prevalence, there is still no cure, and diagnostic methods are extremely subjective, relying on observation of physical motor symptoms and response to treatment protocols. Other neurodegenerative diseases can manifest similar motor symptoms and often too much neuronal damage has occurred before motor symptoms can be observed. The goal of our study is to examine diffusion tensor images (DTI) from Parkinson's and control patients through linear dynamical systems and tensor decomposition methods to generate features for training classification models. Diffusion tensor imaging emphasizes the spread and density of white matter in the brain. We will reduce the dimensionality of these images to allow us to focus on the key features that differentiate PD and control patients. We show through our experiments that these approaches can result in good classification accuracy (90%), and indicate this avenue of research has a promising future.

Index Terms—tensor decomposition, brain imaging, diffusion tensor image, Parkinsons disease

Introduction

Parkinson's Disease

Parkinson's disease (PD) is one of the most common neurodegenerative disorders. The disease mainly affects the motor systems and its symptoms can include shaking, slowness of movement, and reduced fine motor skills. As of 2015 an estimated 6.2 million globally were afflicted with the disease [vos2016]. Its cause is largely unknown and there are some treatments available, but no cure has yet been found. Early diagnosis of PD is a topic of keen interest to diagnosticians and researchers alike. Currently Parkinson's is diagnosed based on the presence of observable motor symptoms and change in symptoms in response to medications that target dopaminergic receptors such as Levodopa [sveinbjornsdottir2016]. The problem with this approach is that it relies on treating symptoms instead of preventing them. Once motor symptoms present, at least 60% of neurons have been affected and there is little likelihood of healing them fully. Additionally early diagnosis will help reduce likelihood of misdiagnosis with other motor neuron diseases.

Parkinsons Progression Markers Initiative Datasets

The Parkinson's Progression Markers Initiative (PPMI) [marek2011] is a clinical study designed to identify PD biomarkers and contribute towards new and better treatments for the disease. The cohort consists of approximately 400 de novo, untreated PD subjects and 200 healthy subjects followed longitudinally for clinical, imaging and biospecimen biomarker assessment. The PPMI data set is a collection of biomarker data collected from a longitudinal study of Parkinson's and control subjects. They have thus far collected DaT (dopamine transporter) scan, MRI (magnetic resonance imaging), fMRI (functional magnetic resonance imaging), and CT (computerized tomography) scan data from several hundred subjects in 6 month intervals. They first began collecting data in 2010, funded by the Michael J.Fox Foundation. The dataset chosen for this paper was PPMI's Diffusion Tensor Imaging (DTI) records. DTI has been shown to be a promising avenue to explore biomarkers in Parkinsonian symptoms and can provide unique insights into brain network connectivity. Moreover, the DTI data was one of PPMI's cleanest and largest datasets and thus expected to be one of the most useful for further analysis. A DTI record is a four-dimensional dataset comprised of a time-series of a three-dimensional imaging sequence of the brain. PPMI's DTIs generally consisted of 65 time slices, each taken approximately five seconds apart. This method tracks movement of water in brain over the discrete time steps, creating a representation of the brain that emphasizes the white matter structures [soares2013].

Existing Work

Parkinson's Disease

A variety of tools currently exist for diagnosis of Parkinson's through pre-motor symptoms. For example Parkinson's seems to measurably affect olfactory sensitivity prior to presenting motor symptoms more than other motor neuron diseases, as illustrated by the University of Pennsylvania Smell Identification Test (UPSIT) [chaudhuri2016]. While there is still more work needed to refine tests like these, it is one example that proves the feasibility of earlier diagnosis of Parkinson's disease. The PPMI holds that discovery of one or more biomarkers for PD is a critical step for developing treatments for the disease. In [chahine2016] a search was conducted of existing PD articles relating to objective biomarkers for PD and found that there are several potential candidates, including biofluids, peripheral tissue, imaging, genetics, and technology based objective motor testing. Dinov et al [dinov2016] explored both model-based and model-free approaches for PD classification and prediction, jointly processing imaging, genetic, clinical, and demographic data. They were able to develop and full data-processing pipeline enabling

* *Corresponding author: rajeswari.a.sivakumar@gmail.com*
‡ *University of Georgia*

modeling of all the data available from PPMI, and found that model-free approaches such as support vector machines (SVM) and K-nearest-neighbor (KNN) outperformed model-based techniques like logistic regression in terms of predicted accuracy. Several of these classifiers generated specificity exceeding 96% when all data available from the dataset was aggregated and used. One interesting finding was a notable increase in accuracy when using group size rebalancing techniques to counteract the effect of cohort sample-size disparities (there are many more patients than control subjects), increasing accuracy in one SVM classifier from 75.9% to 96.3%. Researchers in [baytas2017] recognized the inherent difficulty of using time-series analysis techniques on longitudinal data collected at irregularly-spaced intervals and proposed a new Long-Short Term Memory (LSTM) technique: Time-Aware LSTM (T-LSTM). In [simuni2016] it was found that the subgroup PD classification of tremor dominant (TD) versus postural instability gait disorder dominant (PIGD) has substantial variability, especially in the early stages of diagnosis. For this reason no attempt was made in this paper to include subtype assignment, but only to learn a binary Yes/No PD classification prediction. State-of-the art Parkinson's classification results were reported by [adeli2017] in early 2017 through use of a joint kernel-based feature selection and classification framework. Unlike conventional feature selection techniques, this allowed them to select features that best benefit the classification scheme in the kernel space as opposed to the original input feature space. They analyzed MRI and SPECT data of 538 subjects from the PPMI database and obtained a diagnosis accuracy of 70.5% in MRI generated features and 95.6% in SPECT image generated features. The authors speculated that their non-linear feature selection was the reason for their outperformance of other methods on this non-linear classification problem. Other researchers, [banerjee2016] were able to achieve 98.53% using ensemble learning methods trained on T1 weighted MRI data. However Banerjee used several domain knowledge based feature extraction methods to preprocess their data including image registration, segmentation, and volumetric analysis.

The present research strikes a balance between feature selection and domain knowledge. While our autoregressive model does utilize a basic understanding of relevance of time in diffusion tensor imaging, we do not utilize any other domain specific knowledge to inform our feature extraction. Our hope is to build a generalizable approach that can be applied to other data structured similarly both within and outside the domain of biomedical image analysis. Additionally we want to improve the models being trained without domain specific knowledge on MRI data. This is because MRI is a far less invasive brain imaging method than SPECT imaging which is an X-ray based technique and must be used at a limited frequency. Additionally the multiple MRI modalities offer versatility in examining biological structures.

Tensor and Matrix Decomposition

Matrix decomposition has been used in a variety of computer vision applications in recent years including analysis of facial features. It offers another means of quantifying the features that describe the relationships between values in a 2D space and can be generalized to a variety of applications. The key being that decomposition offers a powerful means of simultaneously evaluating the relationships of values in a 2 or higher dimensional space. In higher dimensional spaces, tensor decomposition is used, where tensors are a generalization of matrices [rabanser2017].

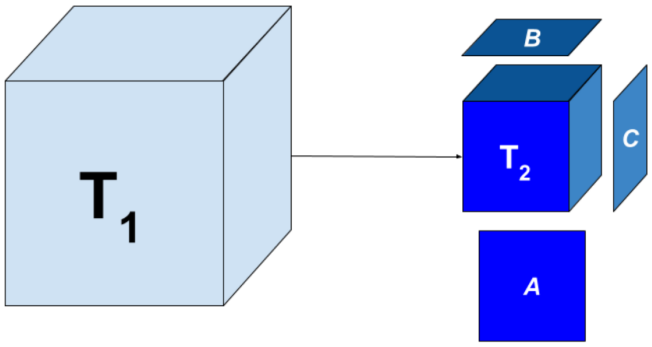


Fig. 1: *Tucker decomposition, visualized.*

Matrix decomposition can be described as a means of separating a matrix into several component matrices whose product would result in the original matrix. For example when solving a system of equations you might approach formulate the problem as:

$$Ax = b$$

where A is a matrix and x and b are vectors. When trying to solve this system of linear equation, we could apply a matrix decompositions operations to the matrix A, to more efficiently solve the system. By finding the products of the of x and b with the one matrix resulting from the decomposition and the inverse of the other, we can solve the system of equations with significantly fewer operations [rabanser2017]. This can be generalized to machine learning applications where increased complexity of models, often result in exponential increases in number of computations. This also affects the applications of new algorithms and pipelines, Those that are too complex and consequently have too many operations become too computationally intensive to be practical to use in some cases. We can choose specific types of decompositions that also allow us to preserve unique information about original matrix while also reducing the size of the matrix. In the case of singular value decomposition we are trying to solve:

$$A = USV^T$$

Where A is the original matrix, of size $m * n$, U is an orthogonal matrix of size $m * n$, S is a diagonal matrix of size $n * n$, and V^T is an orthogonal matrix of size $n * n$. This generalization of the eigendecomposition is useful in compressing matrices without losing information. It will come into play with our final experiment using linear dynamical systems to extract features from the DTIs. Extending the premise of singular value decomposition (SVD) to higher order matrices, or tensors, we come to Tucker decomposition.

Similarly to SVD, Tucker decomposition is used to compress tensors, and can be applied to any tensor of 3 or more dimensions. This is illustrated using a tensor of three dimensions in Figure 1. The resulting core tensor from the decomposition still maintains the same shape and number of dimensions, but each are scaled down to the size specified. We are thus able to use it as means to scale brain images to a set of representative features without breaking down specific regions of interest.

Methods

There are two main experiments conducted. We examine both Tucker tensor decomposition and a linear dynamical systems approach to reduce number of dimensions and scale down diffusion

tensor images. The goal is to evaluate the two approaches for the quality of features extracted. To this end, the final feature vectors produced by each method is then passed on to a random forest classifier, where the accuracy of the final trained model is measured on a classification task to predict control or Parkinson's (PD) group.

The objective is to represent the original DTI as an abstracted tensor that is the product of one of the dimensionality reduction techniques used in each experiment.

Algorithm Selection

To guide our selection of a classifier, we used the python package TPOT [olson2016]. TPOT uses genetic algorithms to iteratively generate, select and evaluate classification pipelines. We evaluated 10 generations of pipelines with population size 100 in each and found that Random Forest classification was most successful as predicting Parkinson's from the generated features. Given the success of random forest classifier, we considered that we might further improve our accuracy by reducing the number of features we used from the generated set. We considered that because we are focused on the differences in a relatively small specific brain regions, only a small number of features would be relevant. To test this theory, we used three different methods to reduce the dimensionality of our feature set to 20 components: linear principle component analysis (PCA), linear discriminant analysis (LDA) and kernel PCA using a radial basis function (RBF).

Experiment I

Using the tensorly package [kossaifi2019], a Tucker decomposition is applied to each brain image. This approach to tensor decomposition was selected because it will produce one core tensor that is representative but scaled down from the original diffusion tensor image. Additionally Tucker decomposition, unlike other forms of tensor decomposition is significantly better at preserving features specific to the tensor being decomposed. Because of this it has applications in compression algorithms. The Tucker decomposition method is chosen in the present study over other tensor decomposition methods to preserve features unique to each brain image it is applied to. This will allow us to scale down each image and focus features and regions of interest in each that are specific to that image. In this experiment we decompose each brain image from a dimension of $(65, 100, 116, 116)$ to $(10, 10, 10, 10)$ to have a continuity in number of features produced.

Experiment II

This experiment focused on breaking down the feature extraction further and evaluate another approach: linear dynamical systems. We scale down each coronal slice in the images and then evaluate the change over time. The reason for scaling down the coronal slices is to allow us to more efficiently build a transition model to represent the flow of water over the time steps of the image. This will allow us to build a three-dimensional representation of the brain from the images that will show the flow of water and the distribution of white matter in the brain. We evaluate the produced transition matrix as features to be applied to the classification pipeline. The nature of the linear dynamical systems allow us to directly model the flow of water via the net change over time in the DTI.

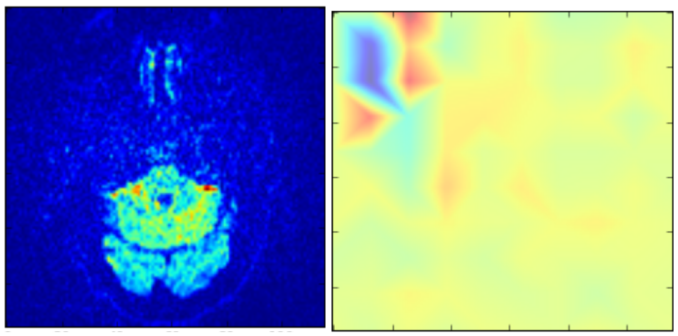


Fig. 2: *(left): Slice from original brain image at a specific time point; (right): Corresponding slice from tensor decomposition output*

Dimensionality Reduction	F-measure	Accuracy
	0.94	0.94
PCA	0.94	0.94
LDA	0.82	0.81
Kernel PCA	0.94	0.94

TABLE 1: *Classification accuracy of features generated from Tucker decomposition after various additional dimensionality reduction techniques are applied*

Results

Experiment I

While we were able to achieve an accuracy of 94% immediately, we were not able to improve on this by further reducing the produced features with various dimensionality reduction methods. In fact it appears that in some cases, such as linear discriminant analysis (LDA), additional dimensionality reduction adversely affects classifier performance. In exploring a slice of the output core tensor at one 'time' point, what we see suggests that the output of the tensor decomposition might be likened to a stack of sliced that focus on the regions of interest in the original image. This is validated by examining several corresponding decomposed core and original slices.

Experiment II

We were able to achieve accuracy of 82% with random forest classifier alone. This outperforms previous benchmarks in training classifiers on synthetic features derived from MR images. Compared to present results, [cole2016] achieved only 70% accuracy at best on synthetic features generated from T1 weighted MRI scans. Furthermore, based on the F-measure scores across the experiment conditions, we can reasonably say that our model is not skewed as a consequence of the uneven distribution of the data. The PPMI

Dimensionality Reduction	F-measure	Accuracy
	0.90	0.82
PCA	0.89	0.81
LDA	0.84	0.74
Kernel PCA	0.93	0.89

TABLE 2: *Classification accuracy of features generated from linear dynamical systems after various additional dimensionality reduction techniques are applied*

data is heavily skewed toward Parkinson's individuals, with a majority of our data set coming from Parkinson's patients (421 subjects) versus controls (213 subjects), which was also addressed by rebalancing the classes by oversampling the control. We intuited that we could speed up model training and improve accuracy by reducing the number of synthetic features we retained. We initially tried linear PCA and LDA to perform the dimensionality reduction. However, these actually hurt performance, resulting in test accuracy of 81% and 74% respectively. Based on this, we considered non-linear dimensionality reduction would be more effective. To this end we used Kernel PCA with RBF kernel, which effectively improved accuracy to 89%.

Discussion

In summary we can conclude that dimensionality reduction is a useful method for extracting meaningful features from brain imaging. Furthermore the impressive performance of these features in machine learning applications indicates that at least some subset of these features strongly correlates with the patient group.

While not explored in this paper, it would be interesting to explore why LDA seemed cause a drop in classifier performance while traditional PCA did not in the tensor decomposition. Furthermore it would be interesting to explore why PCA and LDA both have caused classifier performance decreases with features produced from linear dynamical systems. Specifically it would be interesting to explore the co linearity between the class and features that affect the output features following the LDA treatment. Specifically LDA seems to be stuck producing one strong feature and ignoring the rest.

Additionally it would be interesting to explore the effect of various preprocessing methods to improve out comes and to systematically obscure the data to evaluate which features of the raw pixel data are being hi-lighted by the tensor decomposition and linear dynamical systems steps.

Acknowledgements

Data used in the preparation of this article were obtained from the Parkinson's Progression Markers Initiative (PPMI) database (www.ppmi-info.org/data). For up-to-date information on the study, visit www.ppmi-info.org. PPMI - a public-private partnership - is funded by the Michael J. Fox Foundation for Parkinson's Research and funding partners, including Abbvie, Allergan, Avid, Biogen, BioLegend, Bristol-Mayers Squibb, Colgene, Denali, GE Healthcare, Genentech, GlaxoSmithKline, Lilly, Lundbeck, Merck, Meso Scale Discovery, Pfizer, Piramal, Prevail, Roche, Sanofi Genzyme, Servier, Takeda, TEVA, UCB, Verily, Voyager, and Golub Capital.

We would also like to thanks the reviewers for their feedback and support in preparing this manuscript for publication.

References

[adeli2017] Adeli, E., Wu, G., Saghafi, B., An, L., Shi, F., & Shen, D. (2017). Kernel-based Joint Feature Selection and Max-Margin Classification for Early Diagnosis of Parkinson's Disease.Scientific reports, 7. doi: 10.1038/srep41069

[banerjee2016] Banerjee, M., Okun, M. S., Vaillancourt, D. E., & Vemuri, B. C. (2016). A Method for Automated Classification of Parkinson's Disease Diagnosis Using an Ensemble Average Propagator Template Brain Map Estimated from Diffusion MRI. PloS one, 11(6), e0155764. doi: 10.1371/journal.pone.0155764

[baytas2017] Baytas, I. M., Xiao, C., Zhang, X., Wang, F., Jain, A. K., & Zhou, J. (2017, August). Patient subtyping via time-aware lstm networks. InProceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 65-74). ACM. doi: 10.1145/3097983.3097997

[chahine2016] Chahine, L. M., & Stern, M. B. (2016). Parkinson's Disease Biomarkers: Where Are We and Where Do We Go Next?.Movement Disorders Clinical Practice. doi: 10.1002/mdc3.12545

[chaudhuri2016] Chaudhuri, K. R., Bhidayasiri, R., & van Laar, T. (2016). Unmet needs in Parkinson's disease: New horizons in a changing landscape. Parkinsonism & related disorders, 33, S2-S8. doi: 10.1016/j.parkreldis.2016.11.018

[cole2016] Cole, J. H., Poudel, R. P., Tsagkrasoulis, D., Caan, M. W., Steves, C., Spector, T. D., & Montana, G. (2016, December). Predicting brain age with deep learning from raw imaging data results in a reliable and heritable biomarker. doi: 10.1016/j.neuroimage.2017.07.059

[dinov2016] Dinov, I. D., Heavner, B., Tang, M., Glusman, G., Chard, K., Darcy, M., ... & Foster, I. (2016). Predictive big data analytics: a study of Parkinson's disease using large, complex, heterogeneous, incongruent, multi-source and incomplete observations. PloS one, 11(8), e0157077. doi: 10.1371/journal.pone.0157077

[kossaifi2019] Kossaifi, Jean, et al. "Tensorly: Tensor learning in python." The Journal of Machine Learning Research 20.1 (2019): 925-930. url: http://jmlr.org/papers/v20/18-277.html

[marek2011] Marek, K., Jennings, D., Lasch, S., Siderowf, A., Tanner, C., Simuni, T., ... & Poewe, W. (2011). The parkinson progression marker initiative (PPMI). Progress in neurobiology, 95(4), 629-635. doi: 10.1016/j.pneurobio.2011.09.005

[olson2016] Olson, R. S., Urbanowicz, R. J., Andrews, P. C., Lavender, N. A., & Moore, J. H. (2016, March). Automating biomedical data science through tree-based pipeline optimization. In European Conference on the Applications of Evolutionary Computation (pp. 123-137). Springer, Cham. doi: https://doi.org/10.1007/978-3-030-05318-5_8

[rabanser2017] Rabanser, S., Shchur, O., & Günnemann, S. (2017). Introduction to Tensor Decompositions and their Applications in Machine Learning. arXiv preprint arXiv:1711.10781.

[simuni2016] Simuni, T., Caspell-Garcia, C., Coffey, C., Lasch, S., Tanner, C., Marek, K., & PPMI Investigators. (2016). How stable are Parkinson's disease subtypes in de novo patients: Analysis of the PPMI cohort?.Parkinsonism & related disorders, 28, 62-67. doi: 10.1016/j.parkreldis.2016.04.027

[soares2013] Soares, J. M., Marques, P., Alves, V., & Sousa, N. (2013). A hitchhiker's guide to diffusion tensor imaging. Frontiers in neuroscience, 7. doi: 10.3389/fnins.2013.00031

[sveinbjornsdottir2016] Sveinbjornsdottir, S. (2016). The clinical symptoms of Parkinson's disease. Journal of neurochemistry, 139(S1), 318-324. doi: 10.1111/jnc.13691

[vos2016] Vos, T., Allen, C., Arora, M., Barber, R. M., Bhutta, Z. A., Brown, A., ... & Coggeshall, M. (2016). Global, regional, and national incidence, prevalence, and years lived with disability for 310 diseases and injuries, 1990–2015: a systematic analysis for the Global Burden of Disease Study 2015. The Lancet, 388(10053), 1545-1602. doi: 10.1016/S0140-6736(16)31678-6

PyDDA: A new Pythonic Wind Retrieval Package

Robert Jackson[‡*], Scott Collis[‡], Timothy Lang[‖], Corey Potvin[§¶], Todd Munson[‡]

Abstract—PyDDA is a new community framework aimed at wind retrievals that depends only upon utilities in the SciPy ecosystem such as scipy, numpy, and dask. It can support retrievals of winds using information from weather radar networks constrained by high resolution forecast models over grids that cover thousands of kilometers at kilometer-scale resolution. Unlike past wind retrieval packages, this package can be installed using anaconda for easy installation and, with a focus on ease of use can retrieve winds from gridded radar and model data with just a few lines of code. The package is currently available for download at https://github.com/openradar/PyDDA.

Index Terms—wind, retrieval, hurricane, tornado, radar

Introduction

Three dimensional wind retrievals are important for examining the dynamics that drive severe weather such as tornadoes and hurricanes. In addition, spatial wind retrievals inside severe convection are important for assessing the wind damage they cause. Scanning radars provide the best opportunity for providing three dimensional volumes of winds inside severe weather. However, the retrieval of three dimensional winds from weather radars is a nontrivial task. Given that the radar measures the speed of scatterers in the direction of the radar beam rather than the full wind velocity, retrieving these winds requires more information than the Doppler velocities measured by a single weather radar. Typically, the 3D wind field is retrieved based on constraints with regards to physical laws such as conservation of mass or wind data from other sources such as model reanalyses, wind profilers, and rawinsondes. In particular, atmospheric scientists use two methods to retrieve winds from scanning weather radars. The first method prescribes a strong constraint on the wind field according to the mass continuity equation. The second method is a variational technique that places weak constraints on the wind field by finding the wind field that minimizes a cost function according to deviance from physical laws or from observations ([SPG09], [PSX12]).

Currently existing software for wind retrievals includes software based off of the strong constraint technique such as CEDRIC [MF98] as well as software based off of the weak variational technique such as MultiDop [LSKJ17]. Since CEDRIC uses a strong constraint from mass continuity equation to retrieve winds, the addition of constraints from other data sources is not possible with CEDRIC. Also, while CEDRIC was revolutionary for its time, it is difficult to use as a separate scripting language is the input for the retrieval. While MultiDop is based off of the more customizable 3D variational technique, it is fixed to 2 or 3 radars and is not scalable. Also, Multidop does not support the addition of 3D wind fields from models or other retrievals. Finally, Multidop is a wrapper around a program written in C which introduces issues related to packaging and scalability due to the non-thread-safe nature of the wrapper.

The limitations in current wind retrieval software motivated development of Pythonic Direct Data Assimilation (PyDDA).PyDDA is currently available for download at https://openradarscience.org/PyDDA. PyDDA is entirely written in Python and uses only tools in the Scientific Python ecosystem such as NumPy [vdWCV11], SciPy [JOP+01], and Cartopy [Off15]. This therefore permits the easy installation of PyDDA using pip or anaconda. Given that installation is a major hurdle to using currently existing retrieval software, this makes it easier for those who are not radar scientists to be able to use the software. Unlike currently existing software, a suite of unit tests are built into PyDDA that are executed whenever a user make a contribution to PyDDA, ensuring that the package will function for the user. With regards to ease of use, PyDDA can retrieve winds from multiple radars combined with data from model reanalyses with just a few lines of code. In addition, PyDDA is built upon the Python ARM Radar Toolkit (Py-ART) [HC16]. Since Py-ART is already used by hundreds of users in the radar meteorology community, these users would be able to learn how to use PyDDA easily. Moreover, the open source nature of PyDDA encourages contributions by users for further enhancement. In essence, PyDDA was created with a goal in mind: to make radar wind retrievals more accessible to the scientific community through both ease of installation and use.

This paper will first show the implementation of the variational technique used in PyDDA. After that, this paper shows examples of retrieving and visualizing gridded radar data with PyDDA. Finally, several use cases in severe convection such as Hurricane Florence and a tornado in Sydney, Australia are shown in order to provide examples on how this software can be used by those interested in validating severe weather forecasts and assessing wind damage.

Three dimensional variational (3DVAR) technique

The wind retrieval used by PyDDA is the three dimensional variational technique (3DVAR). 3DVAR retrieves winds by finding the wind vector field $\vec{\mathbf{V}}$ that minimizes the cost function $J(\mathbf{V})$. This

* Corresponding author: rjackson@anl.gov
‡ Argonne National Laboratory, Argonne, IL, USA
‖ NASA Marshall Space Flight Center, Huntsville, AL, USA
§ NOAA/OAR National Severe Storms Laboratory, Norman, OK, USA
¶ School of Meteorology, University of Oklahoma, Norman, OK, USA

Cost function	Basis of constraint
$J_o(\vec{\mathbf{V}})$	Radar observations
$J_c(\vec{\mathbf{V}})$	Mass continuity equation
$J_v(\vec{\mathbf{V}})$	Vertical vorticity equation
$J_m(\vec{\mathbf{V}})$	Model field constraint
$J_b(\vec{\mathbf{V}})$	Background constraint (raw-insonde data)
$J_s(\vec{\mathbf{V}})$	Smoothness constraint

TABLE 1: *List of cost functions implemented in PyDDA.*

Data source	Routine in initialization module
Weather Research and Forecasting (WRF)	`make_background_from_wrf`
High Resolution Rapid Refresh (HRRR)	`make_initialization_from_hrrr`
ERA Interim	`make_initialization_from_era_interim`
Rawinsonde	`make_wind_field_from_profile`
Constant field	`make_constant_wind_field`

TABLE 2: *The differing initializations PyDDA can provide to the user. These initializations are constructed by interpolating the model $J(\vec{V})$ to the analysis grid coordinates.*

cost function is the weighted sum of many different cost functions related to various constraints. The detailed formulas behind these cost functions can be found in [SPG09], [PSX12] as well as in the source code of the `cost_functions` module of PyDDA. The details behind constructing the model constraint are provided in the next section.

The cost function $\vec{\mathbf{V}}$ is then typically expressed as:

$$J(\vec{\mathbf{V}}) = J_o(\vec{\mathbf{V}}) + J_c(\vec{\mathbf{V}}) + J_v(\vec{\mathbf{V}}) + J_m(\vec{\mathbf{V}}) + J_b(\vec{\mathbf{V}}) + J_s(\vec{\mathbf{V}})$$

where each addend is as in Table 1.

The evaluation of $J(\mathbf{V})$ can be done entirely using calls from NumPy and SciPy. For example, evaluating $J_c(\vec{\mathbf{V}}) = \nabla \cdot \vec{\mathbf{V}}$ with an optional anelastic term be reduced to a few NumPy calls. The code that executes these NumPy calls can be found in the Appendix.

Since NumPy can be configured to take advantage of open source mathematics libraries that parallelize the calculation, this also extends the capability of the retrieval to use the available cores on the machine in addition to simplifying the code. Each cost function and its gradient can be expressed in an analytical form using variational calculus, so the addition of more cost functions is possible due to the modular nature of each constraint.

These calculations are then done in order to find the $\vec{\mathbf{V}}$ that minimizes $J(\vec{\mathbf{V}})$. A common technique to minimize $J(\mathbf{V})$ calculates:

$$\vec{\mathbf{V}}_n = \vec{\mathbf{V}_{n-1}} - \alpha(\nabla \vec{\mathbf{V}})$$

for an $\alpha > 0$ until there is convergence to a solution, given that an initial guess $\vec{\mathbf{V}}_0$ is provided. This is called the gradient descent method that finds the minimum by decrementing $\vec{\mathbf{V}}$ in the direction of steepest descent along J. Multidop uses a variant of the gradient descent method, the conjugate gradient descent method, in order to minimize the cost function $J(\vec{\mathbf{V}})$.

However, convergence can be slow for certain cost functions. Therefore, in order to ensure faster convergence, PyDDA uses the limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BGFS) technique that optimizes the gradient descent method by approximating the Hessian from previous iterations. The inverse of the approximate Hessian is then used to find the optimal search direction and α for each retrieval [BLNZ95]. Since there are physically realistic constraints to $\vec{\mathbf{V}}$, the L-BFGS box (L-BFGS-B) variant of this technique can take advantage of this by only using L-BFGS on what the algorithm identifies as free variables, optimizing the retrieval further. In PyDDA, we constrain the solution to ensure that each individual component of $\vec{\mathbf{V}}$ is within a range of $(-100\ m\ s^{-1}, 100\ m\ s^{-1})$.

The L-BFGS-B algorithm is implemented in SciPy. After the initial wind field is provided, PyDDA calls 10 iterations of L-BFGS-B using `scipy.optimize.fmin_l_bfgs_b`. PyDDA will then then test for convergence of a solution by either detecting whether the maximum change in vertical velocity between the current solution and the previous 10 iterations is less than $0.02\ m\ s^{-1}$ or if $\left\|\vec{\mathbf{V}}\right\| < 10^{-3}$, signifying that we have reached a local minimum in $\vec{\mathbf{V}}$. In addition, in order to reduce noise in the retrieved $\vec{\mathbf{V}}$, there are options for the user to use a low pass filter on the retrieval as well as to adjust the smoothness constraint.

Executing the 3DVAR technique with just a few lines of code

With one line of code, one can use the 3DVAR technique to retrieve winds using the `pydda.retrieval.get_dd_wind_field` procedure. If one has a list of Py-ART grids `list_of_grids` that they have loaded and provide $\vec{\mathbf{V}}_0$ into arrays called `u_init`, `v_init`, and `w_init`, retrieval of winds is as easy as

```
winds = pydda.retrieval.get_dd_wind_field(
    list_of_grids, ui, vi, wi)
```

PyDDA even includes an initialization module that will generate example `ui`, `vi`, and `wi` for the user. For example, in order to generate a simple initial wind field of $\vec{\mathbf{V}} = \vec{\mathbf{0}}$ in the shape of any one of the grids in `list_of_grids`, simply do

```
import pydda.initialization as init

ui, vi, wi = init.make_constant_wind_field(
    list_of_grids[0], wind=(0.0, 0.0, 0.0))
```

The user can add their own custom constraints and initializations into PyDDA. Since `pydda.retrieval.get_dd_wind_field` has 3D NumPy arrays as inputs for the initialization, this allows the user to enter in an arbitrary NumPy array with the same shape as the analysis grid as the initialization field.

In addition, PyDDA includes four different initialization routines that will create this field for you from various data sources such as ERA-Interim. Similar to when the constraints are created, the initialization is created by interpolating the original model data from its coordinates to the analysis grid coordinates using nearest-neighbor interpolation. This initialization is then entered in as $\vec{\mathbf{V}}_0$ in the optimization loop.

A similar set of routines exist in in the `constraints` module for creating constraints from model fields. These routines are listed in Table 3. In order to create these constraints, PyDDA will first interpolate the model wind field $\vec{\mathbf{V}}_m$ from the data's original coordinates data into the analysis grid's coordinates using

Data source	Routine in constraints module
Weather Research and Forecasting (WRF)	`make_constraint_from_wrf`
High Resolution Rapid Refresh (HRRR)	`add_hrrr_constraint_to_grid`
ERA Interim	`make_constraint_from_era_interim`

TABLE 3: *The differing model constraints PyDDA can provide to the user. These constraints are constructed by interpolating the model $J(\vec{\mathbf{V}})$ to the analysis grid coordinates.*

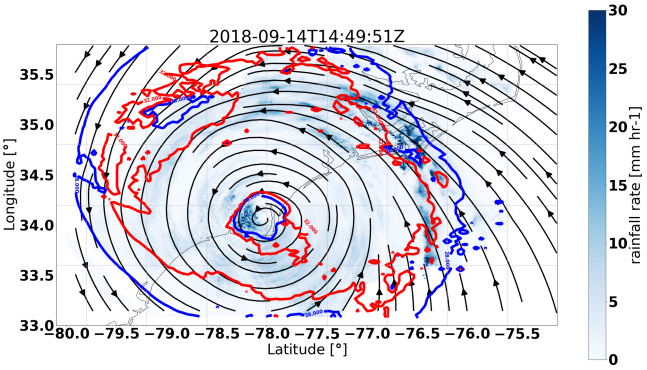


Fig. 1: *An example streamline plot of winds in Hurricane Florence overlaid over radar estimated rainfall rate. The LKTX and KMHX NEXt Generation Radars (NEXRADs) were used to retrieve the winds and rainfall rates. The blue contour represents the region containing gale force winds, while the red contour represents the regions where hurricane force winds are present.*

nearest-neighbor interpolation. After that, for each model, an extra term is added to $J(\vec{\mathbf{V}})$ in the optimization technique. This term corresponds to the sum of the squared error between the $\vec{\mathbf{V}}$ and $\vec{\mathbf{V}}_m$:

$$J_m(\vec{\mathbf{V}}) = c_m \sum_{(i,j,k) \in domain} (v_{ijk} - v_{m,ijk})^2$$

c_m is the weight given to this constraint by the user. The code snippet below will interpolate an HRRR model run to a Py-ART grid called `mygrid`. The `get_dd_wind_field` will then look for the name of the model inside `mygrid` when calculating $J_m(\vec{\mathbf{V}})$.

```
import pydda.constraints as const

# Add HRRR GRIB file
hrrr_path = 'my_hrrr_file.grib'
mygrid = const.add_hrrr_constraint_to_grid(
         mygrid, hrrr_path)
```

The model constraints and retrieval initializations are based off of any 3D field with the same array size and grid specification as the input radar grids. Therefore, these lists can be easily expanded with user routines that interpolate the model or other observational data to the analysis grid.

Visualization module

In addition, PyDDA also supports 3 types of basic visualizations: wind barb plots, quiver plots, and streamline plots. These plots are created using matplotlib and return a matplotlib axis handle so that the user can use matplotlib to make further customizations to the plots. For example, creating a plot of winds on a geographical map with contours overlaid on it such as what is shown in Figure 1 is as simple as:

```
import pyart
import pydda
import cartopy.crs as ccrs

# Load Grids
ltx_grid = pyart.io.read_grid('ltx_grid.nc')
mhx_grid = pyart.io.read_grid('mtx_grid.nc')

# Set up projection and plot of winds
ax = plt.axes(projection=ccrs.PlateCarree())
ax = pydda.vis.plot_horiz_xsection_streamlines_map(
    [ltx_grid, mhx_grid], ax=ax,
    background_field='rainfall_rate', bg_grid_no=-1,
    level=2, vmin=0, vmax=50, show_lobes=False)

# You can add more layers of data that you wish
wind_speed = np.sqrt(ltx_grid.fields["u"]["data"]**2
wind_speed += ltx_grid.fields["v"]["data"]**2)
wind_speed = wind_speed.filled(np.nan)
lons = ltx_grid.point_longitude["data"]
lats = ltx_grid.point_latitude["data"]
cs = ax.contour(
    lons[2], lats[2], wind_speed[2], levels=[28, 32],
    linewidths=8, colors=['b', 'r', 'k'])
plt.clabel(cs, ax=ax, inline=1, fontsize=15)

# Adjust axes properties
ax.set_xticks(np.arange(-80, -75, 0.5))
ax.set_yticks(np.arange(33, 35.8, 0.5))
ax.set_title(ltx_grid.time["units"][-20:])
```

This therefore makes it very easy to create quicklook plots from the data. In addition to horizontal cross sections, PyDDA can also plot wind cross sections in the x-z and y-z planes so that one can view a vertical cross section of winds. Since the `pydda.vis.plot_horiz_xsection_streamlines_map` returns a matplotlib axes handle, it is then possible for the user to customize the plot further to add features such as wind contours as well as adjust the axes limits as shown in the code above.

In addition to streamline plots, PyDDA also supports visualization through quiver plots. Creating a quiver plot from a dataset that looks like Figure 2, in this case a single Doppler retrieval, is as easy as:

```
import pyart
import pydda

Grids = [pyart.io.read_grid('mywinds.nc')]
plt.figure(figsize=(7,7))
pydda.vis.plot_horiz_xsection_quiver(
    Grids, None, 'reflectivity', level=6,
    quiver_spacing_x_km=10.0,
    quiver_spacing_y_km=10.0)
```

In a similar regard, one can also make wind barb plots like the one in Figure 3 using a similar code snippet:

```
import pyart
import pydda

Grids = [pyart.io.read_grid('mywinds.nc')]
plt.figure(figsize=(7,7))
pydda.vis.plot_horiz_xsection_barbs(
    Grids, None, 'reflectivity', level=6,
    barb_spacing_x_km=15.0, barb_spacing_y_km=15.0)
```

More detailed examples on how to visualize wind fields using PyDDA are available at the PyDDA example gallery at https://openradarscience.org/PyDDA/source/auto_examples/index.html.

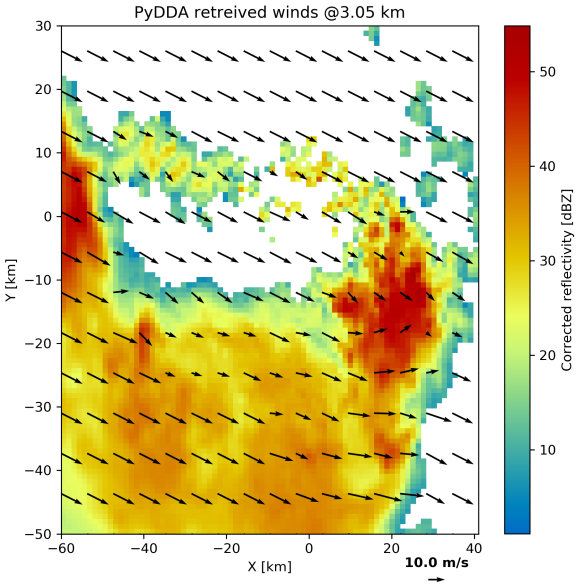


Fig. 2: *An example wind quiver plot from a retrieval from the C-band Polarization Radar, Berrimah radar, and a weather balloon over Darwin on 20 Jan 2006. The background colors represent the radar reflectivity.*

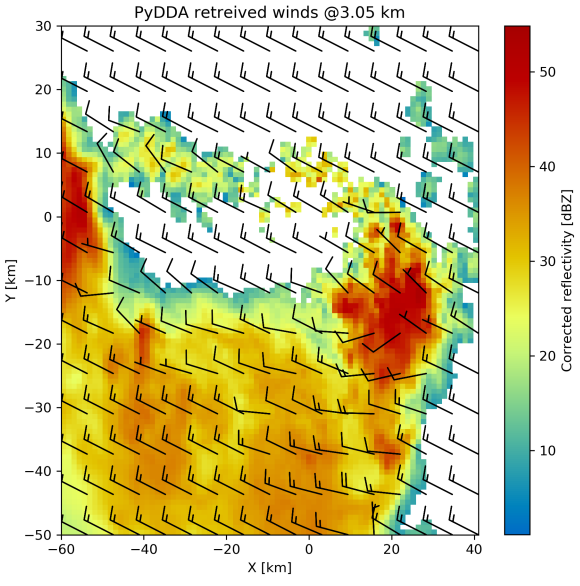


Fig. 3: *As Figure 2, but using wind barbs.*

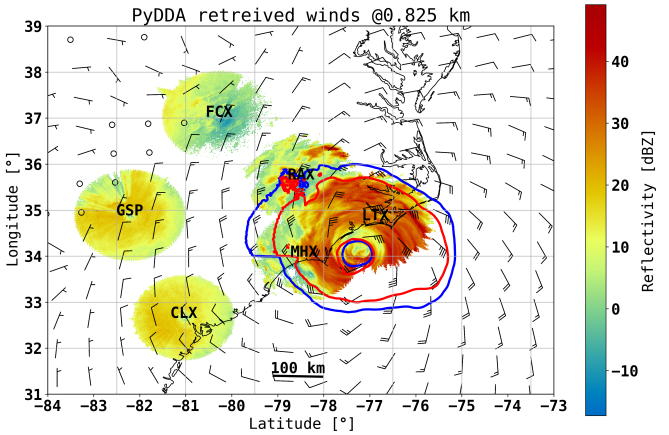


Fig. 4: *A wind barb plot showing the winds retrieved by PyDDA from 6 NEXRADs, the HRRR and the ERA-Interim. The locations of the 6 NEXRADs are marked by their location code. Contours are as in Figure 1.*

Hurricane Florence winds using NEXRAD and HRRR

Another example of the power of PyDDA is its ability to retrieve winds from networks of radars over areas spanning thousands of kilometers with ease. An example retrieval in Hurricane Florence using 2 NEXRAD radars and HRRR was shown in Figure 1. For this grid, the horizontal domain is 300 by 400 km with 1 km grid spacing. While there is already hundreds of kilometers in coverage, not all of the hurricane is covered within the retrieval domain. This therefore motivated a feature in PyDDA to use dask [Das16] to manage retrievals that are too large to execute on one single machine. Figure 4 shows an example of a retrieval from PyDDA using 6 NEXRAD radars combined with the HRRR and ERA-Interim. The total horizontal coverage of the domain in Figure 4 is 1200 km by 1200 km with 1 km spacing. Using a multigrid method that first retrieves the wind field on a coarse grid and then splits the fine grid retrieval into chunks, this technique can use dask to retrieve the wind field in Figure 4 about 30 minutes on 4 nodes with 36-core Intel Broadwell CPUs. The code to retrieve the wind field from many radars and both models is as simple as passing the dask Client instance to the `pydda.get_dd_wind_field_nested` technique. The data and source code for the 2 radar example can be downloaded from https://openradarscience.org/PyDDA/source/auto_examples/index.html.

Given that hurricanes can span hundreds of kilometers and yet have kilometer scale variations in wind speed, having the ability to create such high resolution retrievals is important for those using high resolution wind data for forecast validation and damage assessment. In this example, the coverage of both the tropical storm force and damaging hurricane force winds are examined. Figures 1 and 4 both show kilometer-scale regions of hurricane force winds that may otherwise not have been forecast to occur simply because they are outside of the primary region of damaging winds. This therefore shows the importance of having a high resolution, three dimensional wind retrieval when examining the effects of storm wind damage.

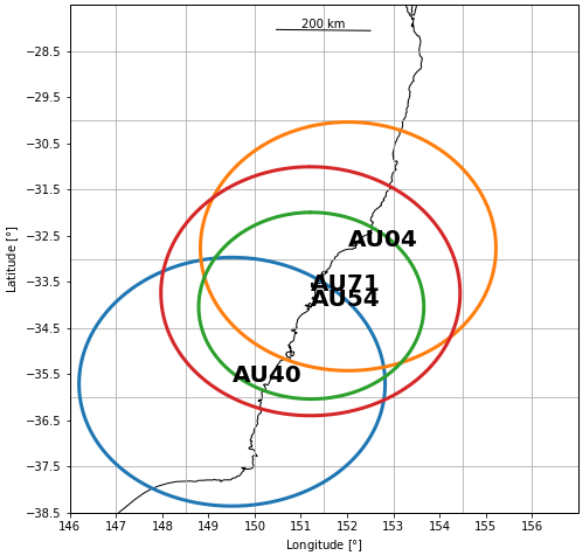


Fig. 5: The locations of the four operational radars operated by the Bureau of Meteorology in the vicinity of Sydney, Australia. The circles represent the maximum unambiguous range of each radar.

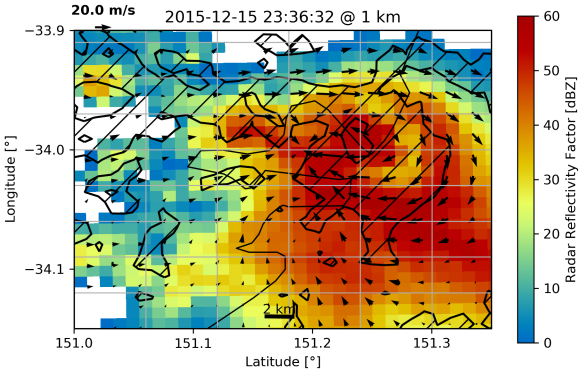


Fig. 6: A quiver plot inside a supercell that spawned a tornado in the vicinity of Sydney, Australia. The area inside the hatched contour represents regions where the updraft velocity is greater than 1 m/s to highlight regions where updrafts are present.

Tornado in Sydney, Australia using 4 radars

In addition to retrieving winds in hurricanes PyDDA can also integrate data from radar networks in order to retrieve the winds inside tornadoes. For example, a network of four scanning radars in the vicinity of Sydney, Australia captured a supercell within the vicinity of Sydney as shown in Figure 5. In this retrieval, a horizontal domain of 350 km by 550 km with 1 km grid spacing was used.

Figure 6 shows the winds retrieved by PyDDA inside this supercell. Using data from the radars, PyDDA is able to provide a complete picture of the rotation inside the supercell and even resolves the updraft in the vicinity of the mesocyclone. Such datasets can be of use for estimating the winds inside a tornado at altitudes as low as 500 m above ground level. This therefore

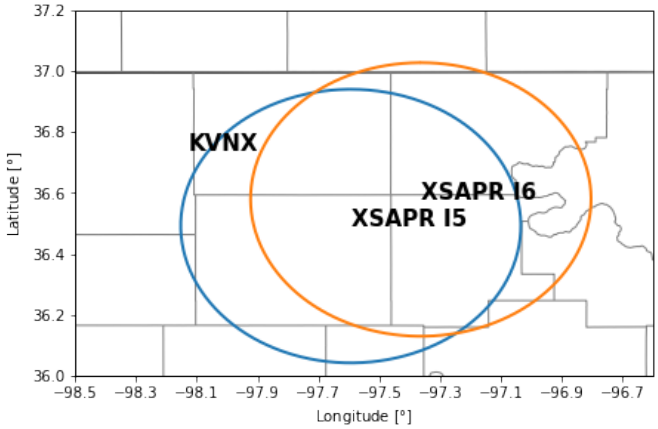


Fig. 7: The locations of the two X-band Scanning Precipitation Radars (XSAPRs) I5 and I6 as well as the KVNX NEXRAD. The two circles represent the maximum unambiguous range of the XSAPR radars. The maximum unambiguous range of KVNX covers the entire figure.

is capable of providing wind datasets that can be used to both provide an estimated wind speed for wind damage assessments as well as for verification of supercell simulations from weather forecasting models. The data and source code for this example is also available at https://openradarscience.org/PyDDA/source/auto_examples/index.html.

Combining winds from 3 scanning radars with HRRR in Oklahoma

A final example shows how easily data from multiple radars and models can be combined together. In this case, we integrate data from three scanning radars whose locations are shown in Figure 7 in the vicinity of the Atmospheric Radiation Measurement (ARM) Southern Great Plains (SGP) site. In this example, the 2 XSAPR radars are at X-band and therefore have lower coverage but greater resolution than the S-band KVNX radar. In addition, the High Resolution Rapid Refresh was used as an additional constraint, with the constraint stronger in regions without radar coverage. The horizontal domain for the retrieval was 100 km by 100 km with 1 km spacing.

Figure 8 shows the resulting wind field of such a retrieval during a case of stratiform rain with embedded convection that occurred over the SGP site on 04 October 2017. Generally, weaker winds and a less organized structure is seen compared to the previous two examples. This would be expected in such conditions. However, this also demonstrates the success in integrating radar data from 3 radars and a high resolution reanalysis to provide the most complete wind retrieval possible. The data and source code for this example is also available at https://openradarscience.org/PyDDA/source/auto_examples/index.html.

Validation

PyDDA utilizes a series of unit tests in order to ensure that quality results are produced with each build of PyDDA. These tests are implemented using pytest. In total, PyDDA currently has 27 tests on the software that test all aspects of the software including the cost functions, optimization loop, and visualizations. For each pull request to the master branch of PyDDA, Travis CI runs this suite of unit tests on

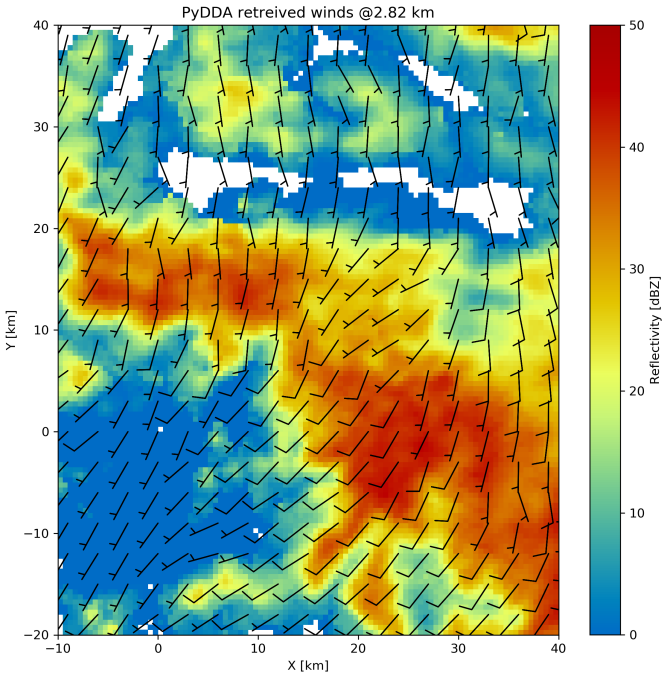


Fig. 8: *A quiver plot of a wind retrieval from 2 XSAPR radars and the KVNX NEXRAD radar in Oklahoma. In addition, the HRRR was used as a constraint. The wind barbs are plotted over the reflectivity derived from the maximum of the reflectivity from the 3 radars.*

the program in order to ensure functionality of the program. Examples of unit tests that are executed by PyDDA are based on expected results from theoretical considerations regarding each cost function. For example, in order to evaluate whether `pydda.cost_functions.calculate_mass_continuity` is working correctly, the tests evaluate this function using a wind field with surface convergence in the center. If the cost function is negative as would be expected, then the unit test passes. Another example evaluates whether the model cost function is working by checking to see if the wind field from the optimization loop converges to the model input if no other data or constraints are specified. In addition, the visualization modules are tested by comparing their results against baseline images to ensure that they are functioning correctly.

Contributor Information

We are currently welcoming contributions from the community into PyDDA. A PyDDA road map demonstrates what kinds of contributions to PyDDA would be useful. As of the writing of this paper, the road map states that the current goals of PyDDA are to implement:

- Support for a greater number of high resolution (LES) models such as CM1 [BF02]
- Support for integrating in data from the Rapid Refresh
- Coarser resolution reanalyses such as the NCEP reanalysis as initializations and constraints.
- Support for individual point analyses, such as those from wind profilers and METARs
- Support for radar data in antenna coordinates
- Improvements in visualizations
- Documentation improvements, including better descriptions in the current English version of the documentation and versions of the documentation in non-English languages.

All contributions to PyDDA will have to be submitted by a pull request to the master branch on https://github.com/openradar/PyDDA. From there, the main developers will examine the pull request to see if unit tests are needed and if the contribution both helps contribute to the goals of the road map and if it passes a suite of unit tests in order to ensure the functionality of PyDDA. In addition, we also require that the user provide documentation for the code they contribute. For the full information on how to make a contribution, go to the contributor's guide at https://openradarscience.org/PyDDA/contributors_guide/index.html.

In addition, for further information about how to use PyDDA, please consult the documentation at https://openradarscience.org/PyDDA.

Acknowledgments

The HRRR data were downloaded from the University of Utah archive [BHL17]. In addition, the authors would like to thank Alain Protat for providing the Sydney tornado wind data. PyDDA was partially supported by the Climate Model Development and Validation Activity of the Department of Energy Office of Science. Dr. Tsengdar Lee of the NASA Weather program provided funds that supported the development of MultiDop, a critical intermediate step toward the development of PyDDA.

Appendix: Mass continuity cost function in Python

This appendix shows an example cost function from PyDDA. The code snippet below shows how the mass continuity cost function can be implemented using NumPy.

```
import numpy as np

def calculate_mass_continuity(
    u, v, w, z, dx, dy, dz, coeff=1500.0, anel=1):
    """
    Calculates the mass continuity cost function by
    taking the divergence
    of the wind field.

    All arrays in the given lists must have the same
    dimensions and represent the same spatial
    coordinates.

    Parameters
    ----------
    u: Float array
        Float array with u component of wind field
    v: Float array
        Float array with v component of wind field
    w: Float array
        Float array with w component of wind field
    dx: float
        Grid spacing in x direction.
    dy: float
        Grid spacing in y direction.
    dz: float
        Grid spacing in z direction.
    z: Float array (1D)
        1D Float array with heights of grid
    coeff: float
        Constant controlling contribution of mass
        continuity to cost function
    anel: int
        = 1 use anelastic approximation, 0=don't
```

```
    Returns
    -------
    J: float
        value of mass continuity cost function
    """
    dudx = np.gradient(u, dx, axis=2)
    dvdy = np.gradient(v, dy, axis=1)
    dwdz = np.gradient(w, dz, axis=0)

    if(anel == 1):
        rho = np.exp(-z/10000.0)
        drho_dz = np.gradient(rho, dz, axis=0)
        anel = w/rho*drho_dz
    else:
        anel = np.zeros(w.shape)
    return coeff*np.sum(
        np.square(dudx + dvdy + dwdz + anel))/2.0
```

REFERENCES

[BF02] George H. Bryan and J. Michael Fritsch. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review*, 130(12):2917–2928, 2002. URL: https://doi.org/10.1175/1520-0493(2002)130<2917:ABSFMN>2.0.CO;2, arXiv:https://doi.org/10.1175/1520-0493(2002)130<2917:ABSFMN>2.0.CO;2, doi:10.1175/1520-0493(2002)130<2917:ABSFMN>2.0.CO;2.

[BHL17] Brian K. Blaylock, John D. Horel, and Samuel T. Liston. Cloud archiving and data mining of high-resolution rapid refresh forecast model output. *Computers and Geosciences*, 109:43 – 50, 2017. URL: http://www.sciencedirect.com/science/article/pii/S0098300417305083, doi:https://doi.org/10.1016/j.cageo.2017.08.005.

[BLNZ95] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995. URL: http://dx.doi.org/10.1137/0916069, doi:10.1137/0916069.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: http://dask.pydata.org.

[HC16] Jonathan Helmus and Scott Collis. The Python ARM Radar Toolkit (Py-ART), a Library for Working with Weather Radar Data in the Python Programming Language. *Journal of Open Research Software*, 4(1), July 2016. URL: http://openresearchsoftware.metajnl.com/articles/10.5334/jors.119/, doi:10.5334/jors.119.

[JOP+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for python, 2001. [Online; accessed <today>]. URL: "http://www.scipy.org/".

[LSKJ17] Timothy Lang, Mario Souto, Shahin Khobahi, and Bobby Jackson. nasa/multidop: Multidop v0.3, October 2017. URL: https://doi.org/10.5281/zenodo.1035904, doi:10.5281/zenodo.1035904.

[MF98] L. Jay Miller and Sherrie M. Fredrick. Custom editing and display of reduced information in cartesian space (cedric) manual. Technical report, National Center for Atmospheric Research, Mesoscale and Microscale Meteorology Division, Boulder, CO., 1998.

[Off15] Met Office. *Cartopy: a cartographic python library with a matplotlib interface*. Exeter, Devon, 2010 - 2015. URL: http://scitools.org.uk/cartopy.

[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. URL: https://doi.org/10.1175/JTECH-D-11-00019.1, arXiv:https://doi.org/10.1175/JTECH-D-11-00019.1, doi:10.1175/JTECH-D-11-00019.1.

[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. URL: https://doi.org/10.1175/2009JTECHA1256.1, arXiv:https://doi.org/10.1175/2009JTECHA1256.1, doi:10.1175/2009JTECHA1256.1.

[vdWCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

Better and faster hyperparameter optimization with Dask

Scott Sievert[‡§*], Tom Augspurger[**], Matthew Rocklin[¶‖]

Abstract—Nearly every machine learning model requires hyperparameters, parameters that the user must specify before training begins and influence model performance. Finding the optimal set of hyperparameters is often a time- and resource-consuming process. A recent breakthrough hyperparameter optimization algorithm, Hyperband finds high performing hyperparameters with minimal training via a principled early stopping scheme for random hyperparameter selection [LJD+18]. This paper will provide an intuitive introduction to Hyperband and explain the implementation in Dask, a Python library that scales Python to larger datasets and more computational resources. The implementation makes adjustments to the Hyperband algorithm to exploit Dask's capabilities and parallel processing. In experiments, the Dask implementation of Hyperband rapidly finds high performing hyperparameters for deep learning models.

Index Terms—distributed computation, hyperparameter optimization, machine learning

Introduction

Training any machine learning pipeline requires data, an untrained model or estimator and "hyperparameters", parameters chosen before training begins that help with cohesion between the model and data. The user needs to specify values for these hyperparameters in order to use the model. A good example is adapting the ridge regression or LASSO to the amount of noise in the data with the regularization parameter [MS75] [Tib96]. Hyperparameter choice verification can not be performed until model training is completed.

Model performance strongly depends on the hyperparameters provided, even for the simple examples above. This gets much more complex when multiple hyperparameters are required. For example, a particular visualization tool, t-SNE requires (at least) three hyperparameters [MH08] and the first section in a study on how to use this tool effectively is titled "Those hyperparameters really matter" [WVJ16].

These hyperparameters need to be specified by the user. There are no good heuristics for determining what the values should be. These values are typically found through a search over possible values through a "cross validation" search where models are scored on unseen holdout data. Even in the simple ridge regression case above, a brute force search is required [MS75]. This brute force search quickly grows infeasible as the number of hyperparameters grow.

Hyperparameter optimization grows more complex as the number of hyperparameters grow, especially because of the frequent interactions between them. A good example of hyperparameter optimization is with deep learning, which has specialized algorithms for handling many data but have difficulty providing basic hyperparameters. For example, the commonly used stochastic gradient descent (SGD) has difficulty with the most basic hyperparameter "learning rate" [Bot10], which is a quick computation with few data but infeasible for many data [MH15].

Contributions

A hyperparameter optimization is required if high performance is desired. In practice, it's expensive and time-consuming for machine learning researchers and practitioners. Ideally, hyperparameter optimization algorithms return high performing models quickly and are simple to use.

Quickly returning quality hyperparameters relies on making decisions about which hyperparameters to devote training time to. This might mean progressively choosing higher-performing hyperparameter values or stopping low-performing models early during training.

Returning this high performing model quickly would lower the expense and/or time barrier to performing hyperparameter optimization. This will allow the user (e.g., a data scientist) to more easily use these algorithms.

This work

- provides an implementation of a particular hyperparameter optimization algorithm, Hyperband [LJD+18] in Dask [Das16], a Python library that provides advanced parallelism. Hyperband returns models with a high validation score with minimal training. A Dask implementation is attractive because Hyperband is amenable to parallelism.
- makes a simple modifications to increase Hyperband's amenability to parallelism.
- provides an simple heuristic to determine the parameters Hyperband requires, which only requires knowing how many examples the model should observe and a rough estimate on how many parameters to sample
- provides validating experiments that illustrate common use cases and explore performance

* *Corresponding author: scott@stsievert.com*
‡ *University of Wisconsin–Madison*
§ *Relevant work performed while interning for Anaconda, Inc.*
** *Anaconda, Inc.*
¶ *NVIDIA*
‖ *Relevant work performed while employed for Anaconda, Inc.*

Hyperband treats computation as a scarce resource[1] and has parallel underpinnings. In the experiments performed with the Dask implementation, Hyperband returns high performing models fairly quickly with a simple heuristic for determining Hyperband's input parameters. The implementation can be found in Dask's machine learning package, Dask-ML[2].

This paper will review other existing work for hyperparameter optimization before detailing the Hyperband implementation in Dask. A realistic set of experiments will be presented to highlight the performance of the Dask implementation before mentioning ideas for future work.

Related work

Hyperparameter optimization

Hyperparameter optimization finds the optimal set of hyperparameters for a given model. These hyperparameters are chosen to maximize performance on unseen data. The hyperparameter optimization process typically looks like

1) Split the dataset into the train dataset and test dataset. The test dataset is reserved for the final model evaluation.
2) Choose hyperparameters
3) Train models with those hyperparameters
4) Score those models with unseen data (a subset of the train dataset typically referred to as the "validation set")
5) Use the best performing hyperparameters to train a model with those hyperparameters on the complete train dataset
6) Score the model on the test dataset. This is the score that is reported.

The rest of this paper will focus on steps 2 and 3, which is where most of the work happens in hyperparameter optimization.

A commonly used method for hyperparameter selection is a random selection of hyperparameters, and is typically followed by training each model to completion. This offers several advantages, including a simple implementation that is very amenable to parallelism. Other benefits include sampling "important parameters" more densely than unimportant parameters [BB12]. This randomized search is implemented in many places, including in Scikit-Learn [PVG+11].

These implementations are by definition *passive* because they do not adapt to previous training. *Adaptive* algorithms can return a higher quality solution with less training by adapting to previous training and choosing which hyperparameter values to evaluate. This is especially useful for difficult hyperparameter optimization problems with many hyperparameters and many values for each hyperparameter.

A popular class of adaptive hyperparameter optimization algorithms are Bayesian algorithms. These algorithms treat the model as a black box and the model scores as an evaluation of that black box. These algorithms have an estimate of the optimal set of hyperparameters and use some probabilistic methods to improve the estimate. The choice of which hyperparameter value to evaluate depends on previous evaluations.

Popular Bayesian searches include sequential model-based algorithm configuration (SMAC) [hut11], tree-structure Parzen estimator (TPE) [STZB+11], and Spearmint [PBBW12]. Many of these are available through the "robust Bayesian optimization" package RoBo [KFMH17] through AutoML[3]. This package also includes Fabolas, a method that takes dataset size as input and allows for some computational control [KFB+16].

Hyperband

Hyperband is a principled early stopping scheme for randomized hyperparameter selection[4] and an adaptive hyperparameter optimization algorithm [LJD+18]. At the most basic level, it partially trains models before stopping models with low scores, then repeats. By default, it stops training the lowest performing 33% of the available models at certain times. This means that the number of models decay over time, and the surviving models have high scores.

Naturally, model quality depends on two factors: the amount of training performed and the values of various hyperparameters. If training time only matters a little, it makes sense to aggressively stop training models. On the flip side, if only training time influences the score, it only makes sense to let all models train for as long as possible and not perform any stopping.

Hyperband sweeps over the relative importance of hyperparameter choice and amount of training. This sweep over training time importance enables a theorem that Hyperband will return a much higher performing model than the randomized search without early stopping returns. This is best characterized by an informal presentation of the main theorem:

Corollary 1. *(informal presentation of [LJD+18, Theorem 5] and surrounding discussion) Assume the loss at iteration k decays like $(1/k)^{1/\alpha}$, and the validation losses ν approximately follow the cumulative distribution function $F(\nu) = (\nu - \nu_*)^{\beta}$ with optimal validation loss ν_* with $\nu - \nu_* \in [0,1]$.*

Higher values of α mean slower convergence, and higher values of β represent more difficult hyperparameter optimization problems because it's harder to obtain a validation loss close to the optimal validation loss ν_*. Taking $\beta > 1$ means the validation losses are not uniformly distributed and higher losses are more common. The commonly used stochastic gradient descent has convergence rates with $\alpha = 2$ [Bot12] [LJD+18, Corollary 6], and gradient descent has convergence rates with $\alpha = 1$ [B+15, Theorem 3.3].

Then for any $T \in \mathbb{N}$, let $\widehat{i_T}$ be the empirically best performing model when models are stopped early according to the infinite horizon Hyperband algorithm when T resources have been used to train models. Then with probability $1 - \delta$, the empirically best performing model $\widehat{i_T}$ has loss

$$\nu_{\widehat{i_T}} \leq \nu_* + c\left(\frac{\overline{\log}(T)^3 \cdot a}{T}\right)^{1/\max(\alpha,\,\beta)}$$

for some constant c and $a = \overline{\log}(\log(T)/\delta)$ where $\overline{\log}(x) = \log(x\log(x))$.

By comparison, finding the best model without the early stopping Hyperband performs (i.e., randomized searches and training until completion) after T resources have been used to train models has loss

$$\nu_{\widehat{i_T}} \leq \nu_* + c\left(\frac{\log(T) \cdot a}{T}\right)^{1/(\alpha+\beta)}$$

For simplicity, only the infinite horizon case is presented though much of the analysis carries over to the practical finite

1. If computation is not a scarce resource, there is little benefit from this algorithm.
2. https://ml.dask.org.
3. https://github.com/automl/
4. In general, Hyperband is a resource-allocation scheme for model selection.

horizon Hyperband.[5] Because of this, it only makes sense to compare the loss when the number of resources used T is large. When this happens, the validation loss of the Hyperband produces $v_{\hat{i}_T}$ decays much faster than the uniform allocation scheme.[6] This shows a definite advantage to performing early stopping on randomized searches.

Li et. al. show that the model Hyperband identifies as the best is identified with a (near) minimal amount of training in Theorem 7 [LJD+18], within log factors of the known lower bound [KCG16].

More relevant work involves combining Bayesian searches and Hyperband, which can be combined by using the Hyperband bracket framework *sequentially* and progressively tuning a Bayesian prior to select parameters for each bracket [FKH18]. This work is also available through AutoML.

There is little to no gain from adaptive searches if the passive search requires little computational effort. Adaptive searches spends choosing which models to evaluate to minimize the computational effort required; if that's not a concern there's not much value the value in any adaptive search is limited.

Dask

Dask provides advanced parallelism for analytics, especially for NumPy, Pandas and Scikit-learn [Das16]. It is familiar to Python users and does not require rewriting code or retraining models to scale to larger datasets or to more machines. It can scale up to clusters or to a massive dataset but also works on laptops and presents the same interface. Dask provides two components:

- Dynamic task scheduling optimized for computation. This low level scheduler provides parallel computation and is optimized for interactive computational workloads.
- "Big Data" collections like parallel arrays, or dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask aims to be familiar and flexible: it aims to parallelize and distribute computation or datasets easily while retaining a task scheduling interface for custom workloads and integration into other projects. It is fast and the scheduler has low overhead. It's implemented in pure Python and can scale from massive datasets to a cluster with thousands of cores to a laptop running single process. In addition, it's designed with interactive computing and provides rapid feedback and diagnostics to aid humans.

Dask's implementation of Hyperband

Combining Dask and Hyperband is a natural fit. Hyperparameter optimization searches often require significant amounts of computation and can involve large datasets. Hyperband is amenable to parallelism, and Dask can scale up to clusters or to massive datasets.

This work focuses on the case when significant computation is required. In these cases, the existing passive hyperparameter optimization algorithms in Dask-ML have limited use because they don't adapt to previous training to reduce the amount of training required.[7]

This section will explain the parallel underpinnings of Hyperband, show the heuristic for Hyperband's inputs and mention a modification to increase amenability to parallelism. Complete documentation of the Dask implementation of Hyperband can be found at https://ml.dask.org/modules/generated/dask_ml.model_selection.HyperbandSearchCV.

Hyperband architecture

There are two levels of parallelism in Hyperband, which result in two for-loops:

- an "embarrassingly parallel" sweep over the different brackets of the training time importance
- each bracket has an early stopping scheme for random search. This means the models are trained independently in parallel. At certain times, training stops on certain models.

The amount of parallelism makes a Dask implementation very attractive. Dask Distributed is required because the computational graph is dynamic and depends on other nodes in the graph.

Of course, the number of models in each bracket decreases over time because Hyperband is an early stopping strategy. This is best illustrated by the algorithm's pseudo-code:

```
from sklearn.base import BaseEstimator

def sha(n_models: int,
        calls: int,
        max_iter: int) -> BaseEstimator:
    """Successive halving algorithm"""
    # (model and params are specified by the user)
    models = [get_model(random_params())
              for _ in range(n_models)]
    while True:
        models = [train(m, calls) for m in models]
        models = top_k(models, k=len(models) // 3)
        calls *= 3
        if len(models) <  3:
            return top_k(models, k=1)

def hyperband(max_iter: int) -> BaseEstimator:
    # Different brackets have different values of
    # "training" and "hyperparameter" importance.
    # => more models means more aggressive pruning
    brackets = [(get_num_models(b, max_iter),
                 get_initial_calls(b, max_iter))
                for b in range(formula(max_iter))]
    if max_iter == 243:  # for example...
        assert brackets == [(81, 3), (34, 9),
                            (15, 27), (8, 81),
                            (5, 243)]
    # Each tuple is (num_models, n_init_calls)
    final_models = [sha(n, r, max_iter)
                    for n, r in brackets]
    return top_k(final_models, k=1)
```

In this pseudo-code, the train set and validation data are hidden. `top_k` returns the `k` best performing models on the validation data and `train` trains a model for a certain number of calls to `partial_fit`.

Each bracket indicates a value in the trade-off between training time and hyperparameter importance, and is specified by the list of tuples in the example above. Each bracket is specified so that the

5. To prove results about the finite horizon algorithm Li et. al. only need the result in Corollary 9 [LJD+18]. In the discussion afterwards they remark that with Corollary 9 they can show a similar result but leave it as an exercise for the reader.

6. This is clear by examining $\log(v_{\hat{i}_T} - v_*)$ for Hyperband and uniform allocation. For Hyperband, the slope approximately decays like $-1/\max(\alpha, \beta)$, much faster than the uniform allocation's approximate slope of $-1/(\alpha+\beta)$.

7. The existing implementation can reduce the computation required when pipelines are used. This is particularly useful when tuning data preprocessing (e.g., with natural language processing). More detail is at https://ml.dask.org/hyper-parameter-search.html.

total number of `partial_fit` calls is approximately the same among different brackets. Then, having many models requires pruning models very aggressively and vice versa with few models. As an example, with `max_iter=243` the least adaptive bracket has 5 models and no pruning. The most adaptive bracket has 81 models and fairly aggressive early stopping schedule.

The exact aggressiveness of the early stopping schedule depends on one optional input to `HyperbandSearchCV`, `aggressiveness`. The default value is 3, which has some mathematical motivation [LJD+18, Section 2.6]. `aggressiveness=4` is likely more suitable for initial exploration when not much is known about the model, data or hyperparameters.

Input parameters

Hyperband is also fairly easy to use. It requires two input parameters:

1) the number of `partial_fit` calls for the best model (via `max_iter`)
2) the number of examples that each `partial_fit` call sees (which is implicit and referred to as `chunks`, which can be the "chunk size" of the Dask array).

These two parameters rely on knowing how long to train the model[8] and having a rough idea on the number of parameters to evaluate. Trying twice as many parameters with the same amount of computation requires halving `chunks` and doubling `max_iter`.

The primary advantage to Hyperband's inputs is that they do not require balancing training time importance and hyperparameter importance.

In comparison, random searches require three inputs:

1) the number of `partial_fit` calls for *every* model (via `max_iter`)
2) how many parameters to try (via `num_params`).
3) the number of examples that each `partial_fit` call sees (which is implicit and referred to as `chunks`, which can be the "chunk size" of the Dask array).

Trying twice as many parameters with the same amount of computation requires doubling `num_params` and halving either `max_iter` or `chunks`, which means every model will see half as many data. Implicitly, a balance between training time and hyperparameter importance is being decided upon. Hyperband has one fewer input because it sweeps over this balance's importance in different brackets.

Dwindling number of models

At first, Hyperband evaluates many models. As time progresses, the number of models decay because Hyperband is an early stopping scheme. This means towards the end of the computation, a few (possibly high-performing) models can be training while most of the computational hardware is free. This is especially a problem when computational resources are not free (e.g., with cloud platforms like Amazon AWS or Google Cloud Platform).

Hyperband is a principled early stopping scheme, but it doesn't protect against at least two common cases:

8. e.g., something in the form "the most trained model should see 100 times the number of examples (aka 100 epochs)"

9. Tolerance (typically via `tol`) is a proxy for `max_iter` because smaller tolerance typically means more iterations are run.

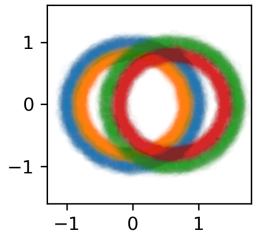


Fig. 1: *The synthetic dataset used as input for the serial simulations. The colors correspond to different class labels. In addition to these two informative dimensions, there are 4 uninformative dimensions with uniformly distributed random noise. There are 60,000 examples in this dataset and 50,000 are used for training.*

1) when models have converged before training completes (i.e., the score stays constant)
2) when models have not converged and poor hyperparameters are chosen (i.e, the scores are not increasing).

Providing a "stop on plateau" scheme will protect against these cases because training will be stopped if a model's score stops increasing [Pre98]. This will require two additional parameters: `patience` to determine how long to wait before stopping a model, and `tol` which determines how much the score should increase.

Hyperband's early stopping is designed to identify the highest performing model with minimal training. Setting `patience` to be high avoids interference with this scheme, protects against both cases above, and errs on the side of giving models more training time. In particular, it also provides a basic early stopping mechanism for the least adaptive bracket of Hyperband.

Serial Simulations

This section is focused on the initial exploration of a model and it's hyperparameters on a personal laptop. This section shows a performance comparison to illustrate the `HyperbandSearchCV`'s utility. This comparison will use a rule-of-thumb to determine the inputs to `HyperbandSearchCV`.

A synthetic dataset is used for a 4 class classification problem on a personal laptop with 4 cores. This makes the hyperparameter selection very serial and the number of `partial_fit` calls or passes through the dataset a good proxy for time. Some detail is mentioned in the appendix with complete details at https://github.com/stsievert/dask-hyperband-comparison.

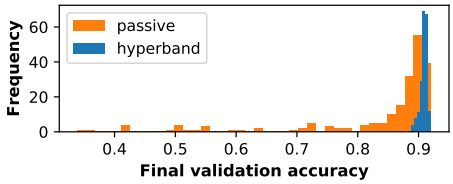
```
from dask_ml.model_selection import train_test_split
X, y = make_4_circles(num=60e3)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=int(10e3))
```

A visualization of this dataset is in Figure 1.

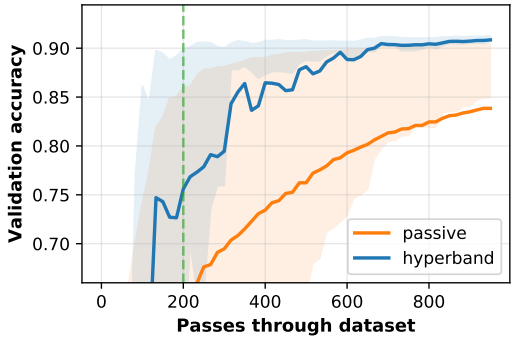
Model architecture & Hyperparameters

Scikit-learn's fully-connected neural network is used, their `MLPClassifier` which has several hyperparameters. Only one affects the architecture of the best model: `hidden_layer_sizes`, which controls the number of layers and number of neurons in each layer.

There are 5 values for the hyperparameter. It is varied so the neural network has 24 neurons but varies the network depth and the width of each layer. Two choices are 12 neurons in 2 layers



(a) *The final validation accuracy over the different runs. Out of the 200 runs, the worst of the `hyperband` runs performs better than 99 of the `passive` runs, and 21 `passive` runs have final validation accuracy less than 70%.*



(b) *The average best score from Hyperband's early stopping scheme (via `hyperband`) and randomized search without any early stopping (via `passive`). The shaded regions correspond to the 25% and 75% percentiles over the different runs. The green dotted line indicates the time required to train 4 models with 4 Dask workers.*

Fig. 2: *In this simulation, each call to `partial_fit` sees about 1/6th of examples in the complete train dataset. Each model completes no more than 50 passes through the data. This experiment includes 200 runs of `hyperband` and `passive` and passive.*

or 6 neurons in four layers. One choice has 12 neurons in the first layer, 6 in the second, and 3 in third and fourth layers.

The other six hyperparameters control finding the best model and do not influence model architecture. 3 of these hyperparameters are continuous and 3 are discrete (of which there are 10 unique combinations). Details are in the appendix. These hyperparameters include the batch size, learning rate (and decay schedule) and a regularization parameter:

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(...)
params = {'batch_size': [32, 64, ..., 512], ...}
print(params.keys())
# dict_keys([
#     "batch_size",  # 5 choices
#     "learning_rate",  # 2 choices
#     "hidden_layer_sizes",  # 5 choices
#     "alpha",  # cnts
#     "power_t",  # cnts
#     "momentum",  # cnts
#     "learning_rate_init"  # cnts
# ])
```

Usage: rule of thumb on `HyperbandSearchCV`'s inputs

`HyperbandSearchCV` only requires two parameters besides the model and data as discussed above: the number of `partial_fit` calls for each model (`max_iter`) and the number of examples each call to `partial_fit` sees (which is implicit via the Dask array chunk size `chunks`). These inputs control how many

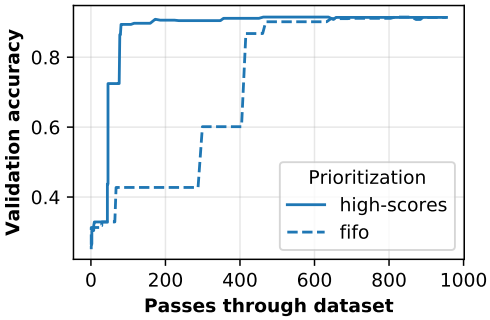


Fig. 3: *A visualization of how the Dask prioritization scheme influences the Hyperband's time to solution. Dask assigns prioritizes training models with higher scores (via `high-scores`). When Dask uses the default priority scheme it fits models in the order they are received by Dask Distributed's scheduler (via `fifo`). Only the prioritization in the figure changes because both `high-scores` and `fifo` have the same hyperparameters, train/validation data, and assign the same internal random state to models. The hyperparameters are chosen from a run in Figure 2b.*

hyperparameter values are considered and how long to train the models.

The values for `max_iter` and `chunks` can be specified by a rule-of-thumb once the number of parameter to be sampled and the number of examples required to be seen by at least one model, `n_examples`. This rule of thumb is:

```
# The rule-of-thumb to determine inputs
max_iter = n_params
chunks = n_examples // n_params
```

In this example, `n_examples = 50 * len(X_train)` and `n_params = 299`. `n_params` is approximately the number of hyperparameter sampled. The value of 299 is chosen to make the Dask array evenly chunked and to sample approximately 4 hyperparameter combinations for unique combination of discrete hyperparameters.

Creation of a `HyperbandSearchCV` object and the Dask array is simple with this:

```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
    model, params,
    max_iter=max_iter, aggressiveness=4)

X_train = da.from_array(X_train, chunks=chunks)
y_train = da.from_array(y_train, chunks=chunks)
search.fit(X_train, y_train)
```

`aggressiveness=4` is chosen because this is my first time optimizing these hyperparameters – I only made one small edit to the hyperparameter search space[10]. With `max_iter`, no model sees more than `n_examples` examples as desired and Hyperband evaluates (approximately) `n_params` hyperparameter combinations[11].

Performance

Two hyperparameter optimizations are compared, Hyperband and random search and is shown in Figure 2b. Recall from above that Hyperband is a principled early stopping scheme for random search. The comparison mirrors that by sampling the same

10. For personal curiosity, I changed total number of neurons to 24 from 20 to allow the `[12, 6, 3, 3]` configuration.

11. Exact specification is available through the `metadata` attribute

Fig. 4: The input and ground truth for the image denoising problem. There are 70,000 images in the output, the original MNIST dataset. For the input, random noise is added to images, and amount of data grows to 350,000 input/output images. Each `partial_fit` *calls sees (about) 20,780 examples and each call to* `score` *uses 66,500 examples for validation.*

hyperparameters[12] and using the same validation set for each run. The results of these simulations are in Figure 2.

Dask provides features that the Hyperband implementation can easily exploit. Dask Distributed supports prioritizing different jobs, so it's simple to prioritize the training of different models based on their most recent score. This will emphasize the more adaptive brackets of Hyperband because they are scored more frequently. Empirically, these are the highest performing brackets of Hyperband [LJD+18, Section 2.3]. This highlights how Dask is useful to Hyperband and is shown in Figure 3.

Dask's priority of training high scoring models works best in very serial environments: priority makes no difference in very parallel environment when every job can be run. In moderately parallel environments the different priorities may lead to longer time to solution because of suboptimal scheduling. To get around this, the worst performing P models all have the same priority for each bracket when there are P Dask workers.

Parallel Experiments

This section will highlight a using a model implemented with a popular deep learning library, and will will leverage Dask's parallelism and investigate how well `HyperbandSearchCV` scales as the number of workers grows from 8 to 32.

The inputs and desired outputs are given in Figure 4. This is an especially difficult problem because the noise variance varies slightly between images. To protect against this, a shallow neural network is used that's slightly more complex than a linear model. This means hyperparameter optimization is not simple.

Specifically, this section will find the best hyperparameters for a model created in PyTorch[13] [PGC+17] (with the wrapper Skorch[14]) for an image denoising task. Again, some detail is mentioned in the appendix and complete details can be found at https://github.com/stsievert/dask-hyperband-comparison.

Model architecture & Hyperparameters

Autoencoders are a type of neural network useful for image denoising. They reduce the dimensionality of the input before expanding to the original dimension, which is similar to a lossy compression. Let's create that model and the images it will denoise:

12. As much as possible – Hyperband evaluates more hyperparameter values. The random search without early stopping evaluates every hyperparameter value Hyperband evaluates.

13. https://pytorch.org

14. https://github.com/skorch-dev/skorch

```
# custom model definition with PyTorch
from autoencoder import Autoencoder
from dask_ml.model_selection import train_test_split
import skorch  # scikit-learn API wrapper for PyTorch

model = skorch.NeuralNetRegressor(Autoencoder, ...)

X, y = noisy_mnist(augment=5)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.05)
```

Of course, this is a neural network so there are many hyperparameters to tune. Only one hyperparameter affects the model architecture: `estimator__activation`, which specifies the activation the neural network should use. This hyperparameter is varied between 4 different choices, all different types of the rectified linear unit (ReLU) [NH10], including the leaky ReLU [MHN13], parametric ReLU [HZRS15a] and exponential linear units (ELU) [CUH15].

The other hyperparameters all control finding the optimal model after the architecture is fixed. These hyperparameters include 3 discrete hyperparameters (with 160 unique combinations) and 3 continuous hyperparameters. Some of these hyperparameters include choices on the optimizer to use (SGD [Bot10] or Adam [KB14]), initialization, regularization and optimizer hyperparameters like learning rate or momentum. Here's a brief summary:

```
params = {'optimizer': ['SGD', 'Adam'], ...}
print(params.keys())
# dict_keys([
#     "optimizer",  # 2 choices
#     "batch_size",  # 5 choices
#     "module__init",  # 4 choices
#     "module__activation",  # 4 choices
#     "optimizer__lr",  # cnts
#     "optimizer__momentum",  # cnts
#     "optimizer__weight_decay"  # cnts
# ])
```

Details are in the appendix.

Usage: plateau specification for non-improving models

`HyperbandSearchCV` supports specifying `patience=True` to make a decision on how long to wait to see if scores stop increasing, as mentioned above. Let's create a `HyperbandSearchCV` object that stops training non-improving models.

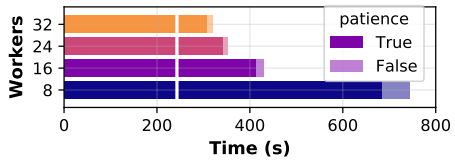
```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
    model, params, max_iter=max_iter, patience=True)
search.fit(X_train, y_train)
```

The current implementation uses `patience=True` to choose a high value of `patience=max_iter // 3`. This is most useful for the least adaptive bracket of Hyperband (which trains a couple models to completion) and mirrors the patience of the second least adaptive bracket in Hyperband.

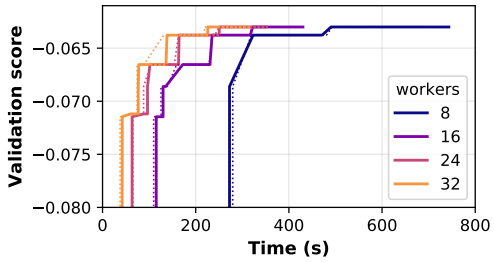
In these experiments, `patience=max_iter // 3` has no effect on performance. If `patience=max_iter // 6` for these experiments, there is a moderate effect on performance (`patience=max_iter // 6` obtains a model with validation loss 0.0637 instead of 0.0630 like `patience=max_iter // 3` and `patience=False`).

Performance

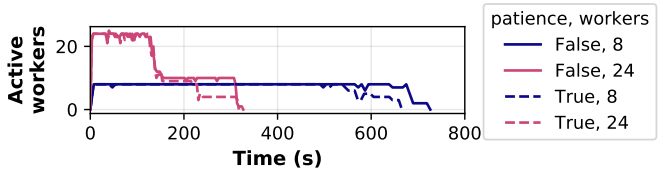
This section will focus on how `HyperbandSearchCV` scales as the number of workers grow.



(a) *The time required to complete the* `HyperbandSearchCV` *search with a different number of workers for different values of* `patience`*. The vertical white line indicates the time required to train one model to completion without any scoring.*



(b) *The time required to obtain a particular validation score (or negative loss) with a different number of Dask workers for* `HyperbandSearchCV` *with* `patience=False` *in the solid line and* `patience=True` *with the dotted line.*



(c) *The effect that specifying* `patience=True` *has on* `HyperbandSearchCV` *for different number of Dask workers.*

Fig. 5: *In these experiments, the models are trained to completion and their history is saved. Simulations are performed with this history that consume 1 second for a* `partial_fit` *call and 1.5 seconds for a* `score` *call. In this simulations, only the number of workers change: the models are static so Hyperband is deterministic. The model trained the longest requires 243 seconds to be fully trained, and additional time for scoring.*

The speedups `HyperbandSearchCV` can achieve begin to saturate between 16 and 24 workers, at least in this experiment as shown in Figure 5b. Figures 5b and 5c show that `HyperbandSearchCV` spends significant amount of time with a low number of workers without improving the score. Luckily, `HyperbandSearchCV` will soon support keyboard interruptions and can exit early if the user desires.

Specifying `patience=True` for `HyperbandSearchCV` has a larger effect on time-to-solution when fewer workers are used as shown in Figure 5a. A stop-on-plateau scheme will have most effect in very serial environments, similar to the priority scheme used by Dask.

Future work

The biggest area for improvement is using another application of the Hyperband algorithm: controlling the dataset size as the scarce resource. This would treat every model as a black box and vary the amount of data provided. This would not require the model to implement `partial_fit` and would only require a `fit` method.

Future work might also include providing an option to further reduce time to solution. This might involve choosing which brackets of `HyperbandSearchCV` to run. Empirically, the best performing brackets are not passive [LJD+18, Section 2.3].

Future work specifically does not include implementing the asynchronous version of successive halving [LJR+18] in Dask. This variant of successive halving is designed to reduce the waiting time in very parallel environments. It does this by stopping a model's training only if it's in the worst performing fraction of models received so far and does not wait for all models to be collected. Dask's advanced task scheduling helps resolves this issue for `HyperbandSearchCV`.

Regardless of these potential improvements, the implementation of Hyperband in Dask-ML allows efficient computation of hyperparameter optimization. The implementation of `HyperbandSearchCV` specifically leverages the abilities of Dask Distributed and can handle distributed datasets.

Appendix

This section expands upon the example given above. Complete details can be found at https://github.com/stsievert/dask-hyperband-comparison.

Serial Simulation

Here are some of the other hyperparameters tuned, alongside descriptions of their default values and the values chosen for tuning.

- `alpha`, a regularization term that can affect generalization. This value defaults to 10^{-4} and is tuned logarithmically between 10^{-6} and 10^{-3}
- `batch_size`, the number of examples used to approximate the gradient at each optimization iteration. This value defaults to 200 and is chosen to be one of $[32, 64, \ldots, 512]$.
- `learning_rate` controls the learning rate decay scheme, either constant or via the `"invscaling"` scheme, which has the learning rate decay like γ_0/t^p where p and γ_0 are also tuned. γ_0 defaults to 10^{-3} and is tuned logarithmically between 10^{-4} and 10^{-2}. p defaults to 0.5 and is tuned between 0.1 and 0.9.
- `momentum`, the amount of momentum to include in Nesterov's momentum [Nes13]. This value is chosen between 0 and 1.

The learning rate scheduler used is not Adam [KB14] because it claims to be most useful without tuning and has reportedly has marginal gain [WRS+17].

Parallel Experiments

Here are some of the other hyperparameters tuned:

- `optimizer`: which optimization method should be used for training? Choices are stochastic gradient descent (SGD) [Bot10] and Adam [KB14]. SGD is chosen with 5/7th probability.
- `estimator__init`: how should the estimator be initialized before training? Choices are Xavier [GB10] and Kaiming [HZRS15b] initialization.
- `batch_size`: how many examples should the optimizer use to approximate the gradient? Choices are $[32, 64, \ldots, 512]$.

- `weight_decay`: how much of a particular type of regularization should the neural net have? Regularization helps control how well the model performs on unseen data. This value is chosen to be zero 1/6th of the time, and if not zero chosen uniformly at random between 10^{-5} and 10^{-3} logarithmically.
- `optimizer__lr`: what learning rate should the optimizer use? This is the most basic hyperparameter for the optimizer. This value is tuned between $10^{-1.5}$ and 10^{1} after some initial tuning.
- `optimizer__momentum`, which is a hyper-parameter for the SGD optimizer to incorporate Nesterov momentum [Nes13]. This value is tuned between 0 and 1.

REFERENCES

[B+15] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–231, 2015.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–281, 2012. URL: http://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187. Springer, Paris, France, August 2010. URL: http://leon.bottou.org/papers/bottou-2010.

[Bot12] Léon Bottou. Stochastic gradient tricks. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks, Tricks of the Trade, Reloaded*, Lecture Notes in Computer Science (LNCS 7700), pages 430–445. Springer, 2012. URL: http://leon.bottou.org/papers/bottou-tricks-2012.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. 80:1437–1446, 10–15 Jul 2018. URL: http://proceedings.mlr.press/v80/falkner18a.html.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[hut11] *Sequential model-based optimization for general algorithm configuration*, volume International Conference on Learning and Intelligent Optimization. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

[HZRS15a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[HZRS15b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KCG16] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *Journal of Machine Learning Research*, 17(1):1–42, 2016. URL: http://jmlr.org/papers/v17/kaufman16a.html.

[KFB+16] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016. URL: https://arxiv.org/abs/1605.07079.

[KFMH17] A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, December 2017. URL: https://github.com/automl/RoBO.

[LJD+18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL: http://jmlr.org/papers/v18/16-558.html.

[LJR+18] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

[MH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. URL: http://jmlr.csail.mit.edu/papers/v9/vandermaaten08a.html.

[MH15] Maren Mahsereci and Philipp Hennig. Probabilistic line searches for stochastic optimization. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 181–189. Curran Associates, Inc., 2015. URL: http://papers.nips.cc/paper/5753-probabilistic-line-searches-for-stochastic-optimization.pdf.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[MS75] Donald W. Marquardt and Ronald D. Snee. Ridge regression in practice. *The American Statistician*, 29(1):3–20, 1975. doi:10.1080/00031305.1975.10479105.

[Nes13] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013. doi:10.1007/978-1-4419-8853-9.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[PBBW12] F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors. *Practical Bayesian Optimization of Machine Learning Algorithms*. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017. URL: https://openreview.net/pdf?id=BJJsrmfCZ.

[Pre98] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998. doi:10.1016/S0893-6080(98)00010-0.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. URL: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[STZB+11] J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors. *Algorithms for Hyper-Parameter Optimization*. Curran Associates, Inc., 2011. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. doi:10.1111/j.2517-6161.1996.tb02080.x.

[WRS+17] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.

[WVJ16] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016. URL: http://distill.pub/2016/misread-tsne, doi:10.23915/distill.00002.

Visualization of Bioinformatics Data with Dash Bio

Shammamah Hossain[‡*]

Abstract—Plotly's Dash is a library that empowers data scientists to create interactive web applications declaratively in Python. Dash Bio is a bioinformatics-oriented suite of components that are compatible with Dash. Visualizations of data that are often found in the field of bioinformatics can now be integrated into Dash applications. We present the Dash Bio suite of components and parts of an auxiliary library that contains tools that parse files from common bioinformatics databases.

Index Terms—visualization, bioinformatics, sequence analysis, Dash

Introduction

The emergent field of bioinformatics is an amalgamation of computer science, statistics, and biology; it has proven itself revolutionary in biomedical research. As scientific techniques in areas such as genomics and proteomics improve, experimentalists in bioinformatics may find themselves needing to interpret large volumes of data. In order to use this data to efficiently provide meaningful solutions to biological problems, it is important to have robust data visualization tools.

Many bioinformaticians have already created analysis and visualization tools with Dash and plotly.py, but only through significant workarounds and modifications made to preexisting graph types. We present an interface to create single-line declarations of charts for complex datasets such as hierarchical clustering and multiple sequence alignment. In addition, we introduce several new chart types, three-dimensional and interactive molecule visualization tools, and components that are specifically related to genomic and proteomic sequences. In a separate library, we present a set of simple parsing scripts that handle some of the most common file types found in bioinformatics-related databases.

This paper outlines the contents of the Dash Bio package. With this package, we hope to impart the powerful data-visualization tools and flexibility of Dash to the flourishing bioinformatics community.

Dash

Plotly's `dash` library provides a declarative Python interface for developing full-stack web applications ("Dash apps"). [Dash] In addition to the main `dash` library, the `dash-html-components` and `dash-core-components` packages comprise the building blocks of a Dash app. `dash-html-components` provides an interface for building the layout of a Dash application that mimics the process of building the layout of a website; `dash-core-components` is a suite of common tools used for interactions with a Dash app (e.g., dropdowns, text inputs, and sliders) and includes a `dcc.Graph` component for interactive graphs made with plotly.py.

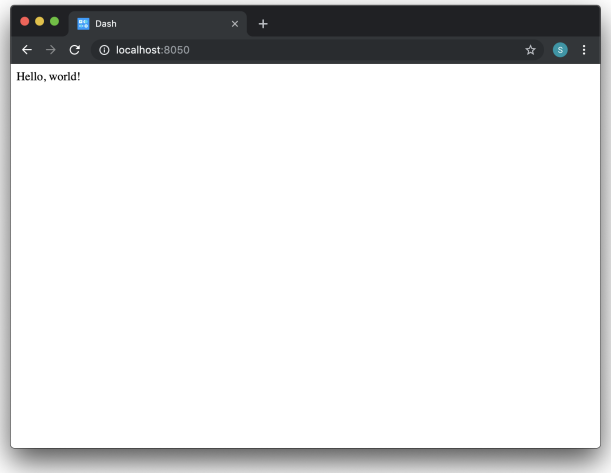


Fig. 1: *A simple Dash application.*

A minimal Dash application that comprises a string on a webpage can be produced with the following code.

```
import dash
import dash_html_components as html

app = dash.Dash()
app.layout = html.Div('Hello, world!')

app.run_server()
```

Upon running the above code, a `localhost` address is specified in the console. Visiting this address in the browser yields a simple webpage that contains the text "Hello, world!" (see Fig. 1).

Interactivity is implemented with callbacks. These allow for reading the values of inputs in the Dash app (e.g., text inputs, dropdowns, and sliders), which can subsequently be used to compute the value of one or more "outputs", i.e., properties of other components in the app. The function that computes the outputs is wrapped in a decorator that specifies the aforementioned inputs and outputs; together, they form a callback. The callback is triggered whenever one of the specified inputs changes in value.

For instance, the `dash_core_components.Input()` component controls the `children` property of a `dash_html_components.Div()` component in the following code.

* *Corresponding author: shammamah@plot.ly*
‡ *Plotly, Inc., 118 - 5555 Avenue de Gaspe, Montreal QC H2T 2A3*

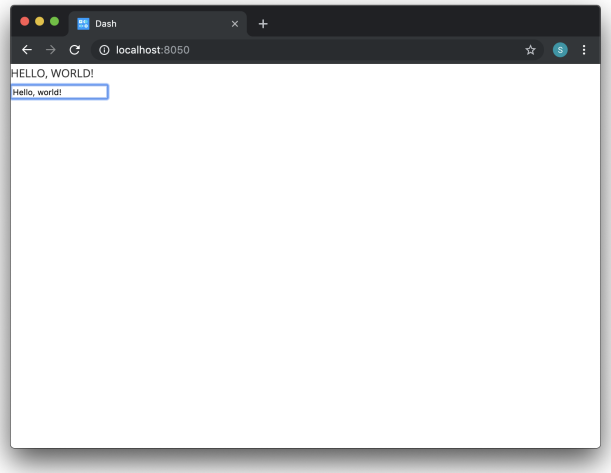


Fig. 2: *A simple Dash application that showcases interactivity. Text that is entered into the input component is converted to uppercase and displayed in the app.*

```
import dash
import dash_html_components as html
import dash_core_components as dcc

app = dash.Dash()
app.layout = html.Div(children=[
     html.Div(id='output-div'),
     dcc.Input(id='text-input')
])

@app.callback(
    dash.dependencies.Output('output-div', 'children'),
    [dash.dependencies.Input('text-input', 'value')]
)
def capitalize_user_input(text):
     return text.upper()

app.run_server()
```

The output of the code is shown in Fig. 2.

React.js and Python

Some of the components in the Dash Bio package are wrappers around pre-existing JavaScript or React libraries. The development process for JavaScript-based components is fairly straightforward; the only thing that needs to be added in many cases is an interface for Dash to access the state of the component and read or write to its properties. This provides an avenue for interactions with the components from within a Dash app.

The package also contains three Python-based components: Clustergram, ManhattanPlot, and VolcanoPlot. Unlike the JavaScript-based components, the Python-based components are essentially functions that return JSON data that is in the format of the `figure` argument for a `dash_core_components.Graph` component.

Dash Bio Components

Dash Bio components fall into one of three categories.

- *Custom chart types:* Specialized chart types that allow for intuitive visualizations of complex data. This category includes Circos, Clustergram, Ideogram, ManhattanPlot, NeedlePlot, and VolcanoPlot.
- *Three-dimensional visualization tools:* Structural diagrams of biomolecules that support a wide variety of user interactions and specifications. This category includes Molecule3dViewer and Speck.
- *Sequence analysis tools:* Interactive and searchable genomic and proteomic sequences, with additional features such as multiple sequence alignment. This category include AlignmentChart, OncoPrint, and SequenceViewer.

The documentation for all of the Dash Bio components, including example code, can be found at https://dash.plot.ly/dash-bio.

Circos

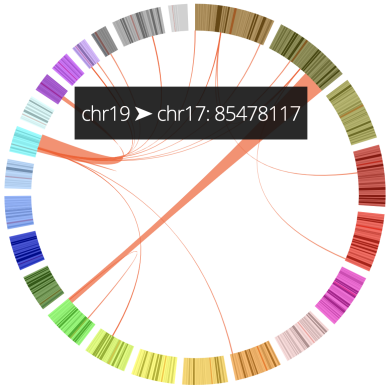


Fig. 3: *A simple Dash Bio Circos component with chords connecting pairs of data points. Data taken from [Ghr] and converted to JSON in the `CircosJS` repository [Circos].*

Circos is a circular graph. It can be used to highlight relationships between, for example, different genes by drawing chords that connect the two (see Fig. 3).

The Dash Bio Circos component is a wrapper of the `CircosJS` [Circos] library, which supports additional graph types like heatmaps, scatter plots, histograms, and stacked charts. Input data to Circos take the form of a dictionary, and are supplied to the `layout` parameter of the component. Additional data, such as a list of chords, are specified in the `tracks` parameter. Multiple tracks can be plotted on the same Circos graph. Hover data and click data on all Circos graph types are captured and are available to Dash apps.

Clustergram

A clustergram is a combination heatmap-dendrogram that is commonly used in gene expression data. The hierarchical clustering that is represented by the dendrograms can be used to identify groups of genes with related expression levels.

The Dash Bio Clustergram component is a Python-based component that uses plotly.py to generate a figure. It takes as input a two-dimensional numpy array of floating-point values. Imputation of missing data and computation of hierarchical clustering both occur within the component itself. Clusters that meet or exceed a user-defined threshold of similarity comprise single traces in the corresponding dendrogram, and can be highlighted with annotations (see Fig. 4).

The user can specify additional parameters to customize the metrics and methods used to compute parts of the clustering, such as the pairwise distance between observations and the linkage

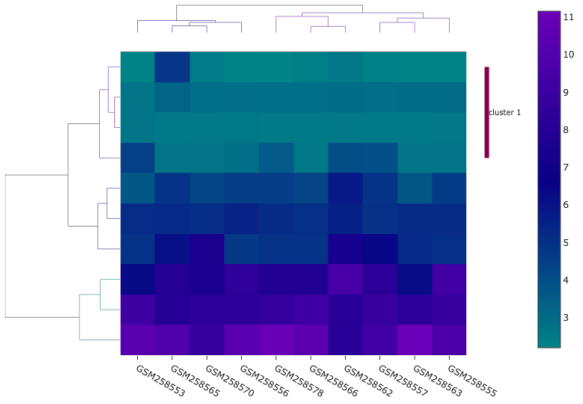


Fig. 4: *A Dash Bio clustergram component displaying hierarchical clustering of gene expression data from two lung cancer subtypes. A cluster from the row dendrogram (displayed to the left of the heatmap) is annotated. Data taken from [KR09].*

matrix. Hover data and click data are accessible from within the Dash app for the heatmap and both dendrograms that are shown in Fig. 4.

Ideogram

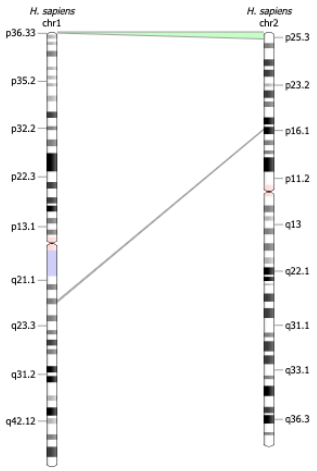


Fig. 5: *A Dash Bio ideogram component demonstrating the homology feature with two human chromosomes. Data taken from the* `ideogram.js` *repository [Ideo].*

An ideogram is a schematic representation of genomic data. Chromosomes are represented as strands, and the locations of specific genes are denoted by bands on the chromosomes.

The Dash Bio Ideogram component is built on top of the `ideogram.js` library [Ideo], and includes features like annotations, histograms, and homology (see Fig. 5). Annotations can be made to different segments of each chromosome and displayed in the form of bands, and relationships between different chromosomes can be highlighted by using the homology feature to connect a region on one chromosome to a region on another (see Fig. 5). Upon hovering over an annotated part of the chromosome, the annotation data is readable from within a Dash app. Additionally, information from the the "brush" feature, which allows the user to highlight a subset of the chromosome, is accessible from within the Dash application. This information includes the starting position and ending position of the brush, as well as the length (in base pairs) of the selection made with the brush.

Manhattan Plot

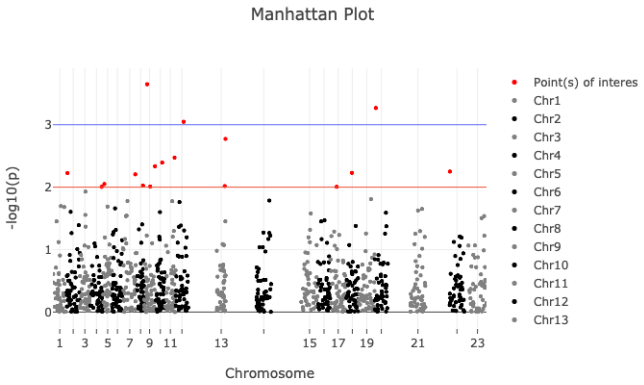


Fig. 6: *A Dash Bio ManhattanPlot component. The threshold level is denoted by the red line; all points of interest are colored red. The purple line is the suggestive line. Data taken from the* `manhattanly` *repository [Man].*

A Manhattan plot is a plot commonly used in genome-wide association studies; it can highlight specific nucleotides that, when changed to a different nucleotide, are associated with certain genetic conditions.

The Dash Bio ManhattanPlot component is built with plotly.py. Input data take the form of a pandas dataframe. The two lines on the plot (see Fig. 6) represent, respectively, the threshold level and the suggestive line.[1] The y-values of these lines can be controlled by the user. Hover data and click data are accessible from within the Dash app.

Needle Plot

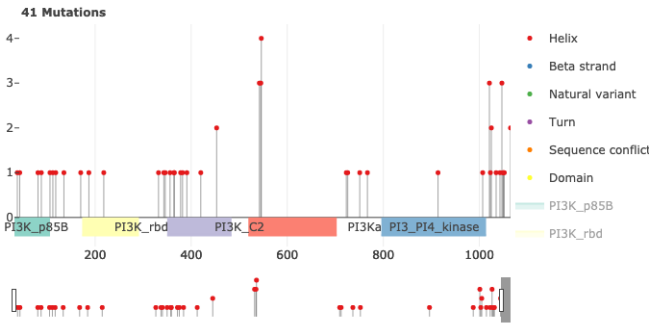


Fig. 7: *A Dash Bio NeedlePlot component that shows the properties of mutations in a genomic strand. Data taken from the* `muts-needle-plot` *repository [Muts].*

A needle plot is a bar plot in which each bar has been replaced with a marker at the top and a line from the x-axis to

1. Information about the meaning of these two lines can be found in [ER15].

the aforementioned marker. Its primary use-case is visualization of dense datasets that would appear too crowded to be interpreted effectively when represented with a bar plot. In bioinformatics, a needle plot may be used to annotate the positions on a genome at which genetic mutations happen (see Fig. 7).

The Dash Bio NeedlePlot component was built using plotly.js. It receives input data as a dictionary. Different colors and marker styles can be used to distinguish different types of mutations, and the domains of specific genes can be demarcated on the plot.

Volcano Plot

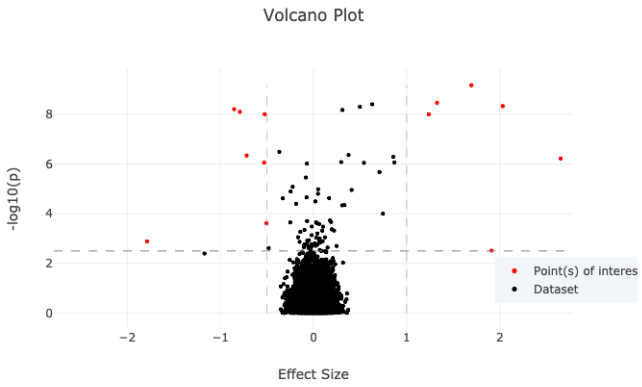


Fig. 8: *A Dash Bio VolcanoPlot component. Points of interest are colored in red, and the effect size and statistical significance thresholds are represented by dashed lines. Data taken from the* `manhattanly` *repository [Man].*

A volcano plot is a plot used to concurrently display the statistical significance and a defined "effect size" (e.g., the fold change[2]) of a dataset. This type of plot is incredibly useful when visualizing a large number of data points that represent replicate data; it facilitates identification of data that simultaneously have statistical significance and a large effect.

The Dash Bio VolcanoPlot component was built using plotly.py. It takes a pandas dataframe as input data. Lines that represent the threshold for effect size (both positive and negative) and a threshold for statistical significance can be defined by the user (see Fig. 8). Hover data and click data are accessible from within the Dash app.

Molecule 3D Viewer

The Dash Bio Molecule3dViewer component was built on top of the `molecule-3d-for-react` [Mol3D] library. Its purpose is to display molecular structures. These types of visualizations can show the shapes of proteins and provide insight into the way that they bind to other molecules. This renders them invaluable when communicating the mechanics of biomolecular processes.

Molecule3dViewer receives input data as a dictionary which specifies the layout and style of each atom in the molecule. It can render molecules in a variety of styles, such as ribbon diagrams, and allows for mouse-click selection of specific atoms or residues (see Fig. 9) that can be read from or written to within a Dash app.

Speck

The Dash Bio Speck component is a WebGL-based 3D renderer that is built on top of `Speck` [Speck]. It uses techniques like

2. This refers to the ratio of a measurement to its preceding measurement.

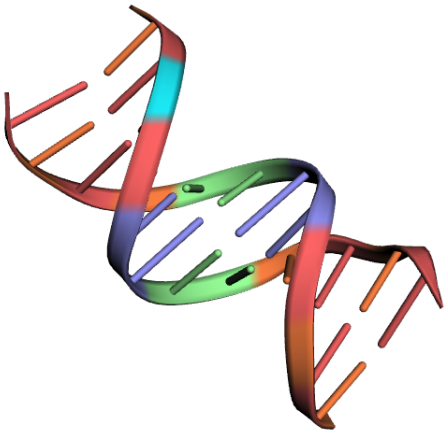


Fig. 9: *A Dash Bio Molecule3DViewer component displaying the ribbon structure of a section of DNA. A selected residue is highlighted in cyan. Structural data taken from the Protein Data Bank [1bna].*

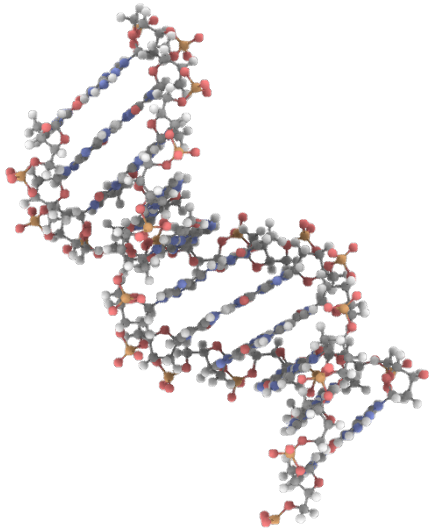


Fig. 10: *A Dash Bio Speck component displaying the atomic structure of a strand of DNA in a ball-and-stick representation. Ambient occlusion is used to provide realistic shading on the atoms. Structural data taken from the* `Speck` *repository [Speck].*

ambient occlusion and outlines to provide a rich view of molecular structures (see Fig.).

The Dash Bio Speck component receives input data as a dictionary that contains, for each atom, the atomic symbol and the position in space (given as x, y, and z coordinates). Parameters related to the rendering of the molecule, such as the atom sizes, levels of ambient occlusion, and outlines, can optionally be specified in another dictionary supplied as an argument.

Alignment Chart

An alignment chart is a tool for viewing multiple sequence alignment. Multiple related sequences of nucleotides or amino acids (e.g., the amino acid sequences of proteins from different

Fig. 11: *A Dash Bio AlignmentChart component displaying the P53 protein's amino acid sequences from different organisms. A conservation barplot is displayed on top, and the bottom row of the heatmap contains the consensus sequence. Data taken from UniProt [UniP].*

organisms that appear to serve the same function) are displayed in the chart to show their similarities.

The Dash Bio AlignmentChart component is built on top of `react-alignment-viewer` [Align]. It takes a FASTA file as input and computes the alignment. It can optionally display a barplot that represents the level of conservation of a particular amino acid or nucleotide across each sequence defined in the input file (see Fig. 11). Hover data and click data are accessible from within the Dash app.

Onco Print

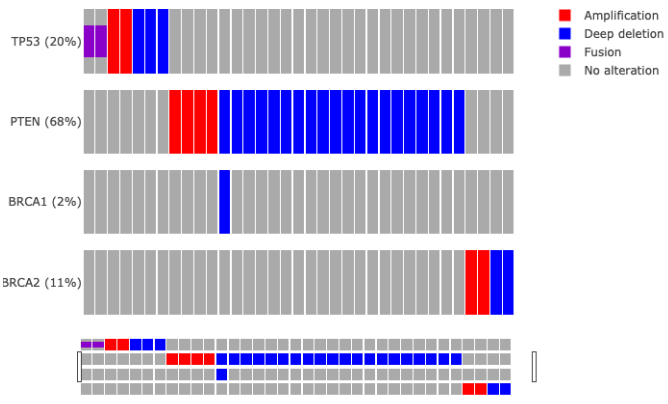


Fig. 12: *A Dash Bio OncoPrint component that shows mutation events for the genomic sequences that encode different proteins. Data taken from cBioPortal [cBio], [cBio2].*

An OncoPrint graph is a type of heatmap that facilitates the visualization of multiple genomic alteration events (see Fig. 12).

The Dash Bio OncoPrint component is built on top of `react-oncoprint` [Onco]. Input data for the component takes the form of a list of dictionaries that each define a sample, gene, alteration, and mutation type. Hover data and click data are accessible from within the Dash app.

Sequence Viewer

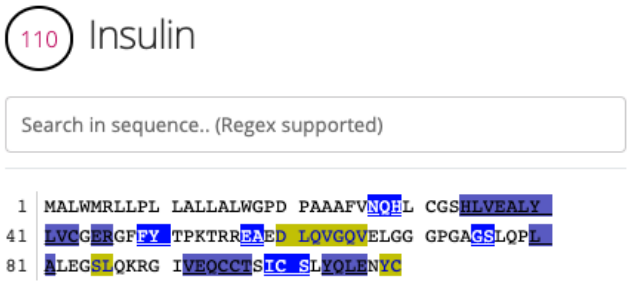


Fig. 13: *A Dash Bio SequenceViewer component that is showing the amino acid sequence for insulin. A coverage has been applied to the sequence to emphasize subsequences of amino acids that form certain structures, like alpha helices or beta sheets. Data taken from NeXtProt [nXP].*

The Dash Bio SequenceViewer component is a simple tool that allows for annotating genomic or proteomic sequences. It is based on the `react-sequence-viewer` library [SeqV].

It includes a search function that allows the user to search the sequence using regular expressions. In addition, the sequence can be annotated using a selection defined by a starting point, an end point, and a color, or a coverage that can encode additional information that is revealed once a subsequence is clicked. The selection and coverage are available for reading from and writing to in the Dash app, and the mouse selection and search results are also accessible.

File Parsers

The `dash-bio-utils` package was developed in tandem with the `dash-bio` package. It contains parsers for common filetypes used in bioinformatics analyses. The parsers in the package translate the data encoded in those files to inputs that are compatible with Dash Bio components.

FASTA data

FASTA files are commonly used to represent one or more genomic or proteomic sequences. Each sequence may be preceded by a line starting with the > character which contains information about the sequence, such as the name of the gene or organism; this is the description of the sequence. Sections of the description are separated with pipes (|).

The `protein_reader` file in the `dash-bio-utils` package accepts a file path to, or a string representation of, a FASTA file, and returns a dictionary that contains the sequence and any metadata that are specified in the file. `SeqIO` from the `Biopython` [BioP] package was used to extract all of the sequences from the file into a list of dictionaries, each of which contained the sequence description and the sequence itself, both in string format.

Different databases (e.g., neXtProt, GenBank, and SWISS-PROT) encode the sequence description metadata in different ways. The database from which a FASTA file is retrieved is

specified in the first line. In the `protein_reader` file, the code for the database is translated into the information that is encoded in the first line for that particular database. [NCBI]

From there, string splitting (or, if necessary, regex) is used on the description line of the file to generate a dictionary of the sequence metadata.

This parser enables quick access to all of the information contained in a FASTA file, which in turn can make the information more human-readable. This is a feature that supplements the ease-of-use of the `dash-bio` package.

For instance, in the code snippet below, the parser is used on a string with the contents of a FASTA file for the albumin protein [nXP]:

```
>>> from dash_bio_utils import protein_reader as pr
>>> fasta_string = \
'''>nxp|NX_P02768-1|ALB|Serum albumin|Iso 1

MKWVTFISLLFLFSSAYSRGVFRRDAHKSEVAHRFKDLGEENFKALVLIAF
AQYLQQCPFEDHVKLVNEVTEFAKTCVADESAENCDKSLHTLFGDKLCTVA
TLRETYGEMADCCAKQEPERNECFLQHKDDNPNLPRLVRPEVDVMCTAFHD
NEETFLKKYLYEIARRHPYFYAPELLFFAKRYKAAFTECCQAADKAACLLP
KLDELRDEGKASSAKQRLKCASLQKFGERAFKAWAVARLSQRFPKAEFAEV
SKLVTDLTKVHTECCHGDLLECADDRADLAKYICENQDSISSKLKECCEKP
LLEKSHCIAEVENDEMPADLPSLAADFVESKDVCKNYAEAKDVFLGMFLYE
YARRHPDYSVVLLLRLAKTYETTLEKCCAAADPHECYAKVFDEFKPLVEEP
QNLIKQNCELFEQLGEYKFQNALLVRYTKKVPQVSTPTLVEVSRNLGKVGS
KCCKHPEAKRMPCAEDYLSVVLNQLCVLHEKTPVSDRVTKCCTESLVNRRP
CFSALEVDETYVPKEFNAETFTFHADICTLSEKERQIKKQTALVELVKHKP
KATKEQLKAVMDDFAAFVEKCCKADDKETCFAEEGKKLVAASQAALGL'''
>>> albumin = pr.read_fasta(
...     data_string=fasta_string
... )[0]
>>> albumin['description']
{'identifier': 'NX_P02768-1',
 'gene name': 'ALB',
 'protein name': 'Serum albumin',
 'isoform name': 'Iso 1'}
>>> albumin['sequence'][:10]
'MKWVTFISLL'
```

Gene Expression Data

Gene expression data take the form of two-dimensional arrays that measure expression levels for sets of genes under varying conditions.

A common format that is used to represent gene expression data is the SOFT format. These files can be found in large databases such as the Gene Expression Omnibus (GEO), [GEO] which contains gene expression data from thousands of experiments. SOFT files contain the expression data, as well as descriptive information pertaining to the specific genes and conditions that are in the dataset.

The `gene_expression_reader` file in the `dash-bio-utils` package accepts a path to, or a string representation of, a SOFT file or TSV file containing gene expression data. It can parse the contents of SOFT and TSV files, and return the numerical data and metadata that they contain. In addition, selection of a subset of the data (given by lists of selected rows and selected columns supplied to the parser) can be returned.

The `GEOparse` package [GEOP] was used to extract the numeric gene expression data to a `pandas` dataframe, in addition to the metadata, in SOFT files:

```
geo_file = gp.get_GEO(
    filepath=filepath,
    geotype='GDS'
)
df = geo_file.table
```

`pandas` was used to do the same with TSV files:

```
df = pd.read_csv(
    filepath, sep='\t'
)
```

Both file parsers by default return a tuple comprising the file metadata, all of the row names, and all of the column names.

If the parameter `return_filtered_data` is set to `True`, the parameters `rows` and `columns` (lists that contain the names of, respectively, the selected rows and selected columns) must be specified. The dataframe `df` is then filtered according to these selections, and a two-dimensional `numpy` array containing the filtered data is returned.

In the case of SOFT files, there is additional information about subsets of the dataset (e.g., the expression data that are recorded with and without inducing a particular gene). This information becomes another element in the tuple.

In the code snippet below, the parser is used to extract information from a dataset related to the miR-221 RNA molecule [miR]:

```
>>> from dash_bio_utils import gene_expression_reader
>>> data = gene_expression_reader.read_soft_file(
...     filepath='GDS5373.soft'
... )
>>> data[0]
{'title': [
'''miR-221 expression effect on prostate cancer
 cell line'''
],
'description': [
'''Analysis of PC-3 prostate cancer cells
   expressing pre-miR-221. miR-221 is frequently
   downregulated in primary prostate cancer.
   Results provide insight into the role of
   miR-221 in the pathogenesis of prostate
   cancer.'''
],
'type': ['Expression profiling by array'],
'pubmed_id': ['24607843'],
'platform': ['GPL570'],
'platform_organism': ['Homo sapiens'],
'platform_technology_type': ['in situ oligonucleotide'],
'feature_count': ['54675'],
'sample_organism': ['Homo sapiens'],
'sample_type': ['RNA'],
'channel_count': ['1'],
'sample_count': ['4'],
'value_type': ['count'],
'reference_series': ['GSE45627'],
'order': ['none'],
'update_date': ['Nov 03 2014']}
>>> data[1]
{'GDS5373_1': {'dataset_id': ['GDS5373'],
               'description': ['miR-122 expression'],
               'sample_id': ['GSM1110879,GSM1110880'],
               'type': ['protocol']},
'GDS5373_2': {'dataset_id': ['GDS5373'],
              'description': ['control'],
              'sample_id': ['GSM1110881,GSM1110882'],
              'type': ['protocol']}}
>>> data[2][:10]
['1007_s_at', '1053_at', '117_at', '121_at',
 '1255_g_at', '1294_at', '1316_at', '1320_at',
 '1405_i_at', '1431_at']
>>> data[3]
['GSM1110879', 'GSM1110880', 'GSM1110881', 'GSM1110882']
>>> selected = gene_expression_reader.read_soft_file(
...     filepath='GDS5373.soft',
...     rows=['1255_g_at', '1316_at'],
...     columns=['GSM1110879', 'GSM1110881'],
...     return_filtered_data=True
... )
>>> selected
```

```
array([[22.7604, 23.0321],
       [21.416 , 21.0107]])
```

Molecule Structural Data

The Protein Data Bank (PDB) [PDB] is a database of files that describe macromolecular structural data. All of the files on PDB are in the PDB format.

In the `dash_bio_utils` package, the `create_data` function in `pdb_parser` generates a JSON string from the contents of a specified PDB file. This string contains information about the atoms and the bonds in the molecular structure.

The PDB format is standardized; properties of each atom such as its position in space and the chain and residue to which it belongs are found within specific column indices for each row. [PdbF] `pdb_parser` uses this information to parse each line, and creates a list of dictionaries, each of which contains information about the aforementioned properties for each atom in the PDB file.

The `parmed` library [Par] was used to read the bond information from the PDB file. Using the bond information from `parmed`, a list of dictionaries is created; each dictionary contains the indices of the pair of atoms that form a bond.

In the code snippet below, this parser is used to extract data from a PDB file that contains structural information for a small section of DNA: [1bna]

```
>>> import json
>>> from dash_bio_utils import pdb_parser
>>> pdb_string = pdb_parser.create_data('1bna.pdb')
>>> pdb_1bna = json.loads(pdb_string)
>>> pdb_1bna['atoms'][:3]
[{'name': "O5'", 'chain': 'A',
'positions': [18.935, 34.195, 25.617],
'residue_index': 1, 'element': 'O',
'residue_name': 'DC1', 'serial': 0},
{'name': "C5'", 'chain': 'A',
'positions': [19.13, 33.921, 24.219],
'residue_index': 1, 'element': 'C',
'residue_name': 'DC1', 'serial': 1},
{'name': "C4'", 'chain': 'A',
'positions': [19.961, 32.668, 24.1],
'residue_index': 1, 'element': 'C',
'residue_name': 'DC1', 'serial': 2}]
>>> pdb_1bna['bonds'][:3]
[{'atom2_index': 0, 'atom1_index': 1},
{'atom2_index': 1, 'atom1_index': 2},
{'atom2_index': 2, 'atom1_index': 3}]
```

Comparisons with Existing Tools

GenomeDiagram

The GenomeDiagram package [Geno] provides a way to visualize comparative genomics data in a circular format (see Fig. 14); supported chart types include line charts, bar graphs, and heatmaps.

GenomeDiagram can additionally export high-quality vector diagrams of the charts that are generated, which can in turn be used in research papers. It can be used in conjunction with the BioPython module to interface with GenBank.

GenomeDiagram shares many similarities with the Circos component; both are circular representations of genomic data, and both support multiple "tracks", or traces, of multiple chart types. The key difference between the two, and the advantage of Dash Circos, is flexibility and interactivity; Dash Circos supports click and hover interactions, and GenomeDiagram does not.

Furthermore, Dash Circos can be interactively modified with respect to the data that are displayed, as well as the appearance of the graph itself. This allows for the implementation of many useful

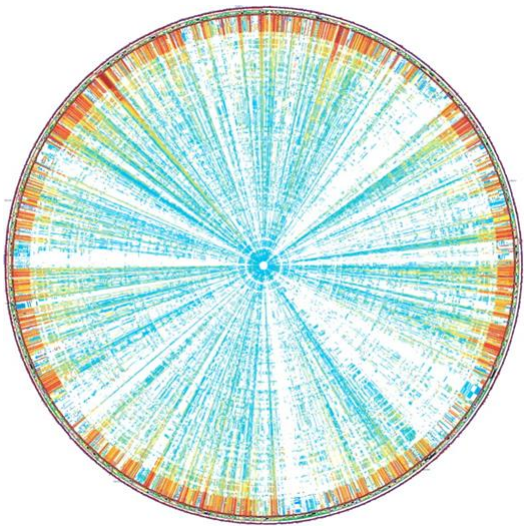


Fig. 14: *An example of a circular diagram that can be generated with GenomeDiagram. Source: [Geno]*

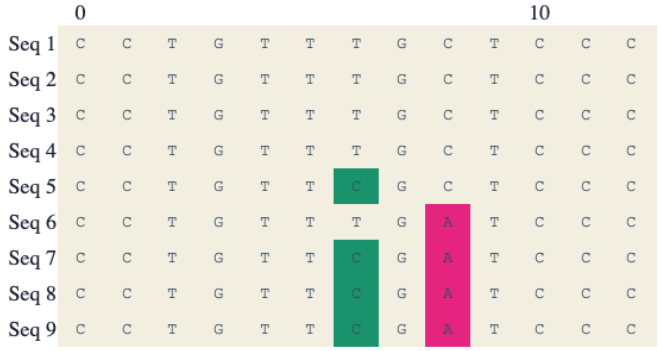


Fig. 15: *Part of a multiple sequence alignment displayed as a Plotly heatmap. Source: [JCViz]*

functions, such as cross-filtering. Instead of needing to re-create the image every time a change is made, updates to the Circos component are reflected immediately within a Dash app.

Plotly.py

Plotly's Python plotting library has been used to create charts that are visually similar to those that are used to display certain types of bioinformatics data [JCViz]. For instance, a sequence alignment viewer can be created with a Plotly heatmap (see Fig. 15).

The Dash Bio AlignmentViewer component applies a similar approach; the React.js component uses a plotly.js heatmap to display the alignment. However, the API of AlignmentViewer differs from that of the Plotly.py heatmap. The latter requires the user to define properties of the graph that don't have anything to do with the alignment itself. Annotations must be specified, as well as a custom heatmap colorscale in which the values correspond to bases and not percentiles of a dataset. It also requires preprocessing of the FASTA data, and translation into a format that can be fit into the parameters of a Plotly heatmap.

In contrast, AlignmentViewer includes support for information that is specific to multiple sequence alignment. The gap and conservation, for instance, can be optionally included as barplots; the method of conservation can also be changed, and the consensus

sequence can be displayed on the chart. Data in the form of FASTA files can be used as input to the component without any further processing required. This allows for the programmer to more easily interact with the component, as it removes the need to restructure data to fit a specific format for visualization.

Limitations and Future Directions

File Formats

Currently, the `dash_bio_utils` package only supports specific data file formats (namely, PDB, FASTA, and SOFT). Additionally, most of the components require JSON data as input; this file format is not typically provided in datasets or studies. Future developments to the package should therefore include processing for other important file formats, such as SAM/BAM/BAI for sequence alignment, or Genbank files (.gb).

Conclusion

The Dash Bio component suite facilitates visualization of common types of datasets that are collected and analyzed in the field of bioinformatics. It remains consistent with the declarative nature of Plotly's Dash, and permits users to create interactive and responsive web applications that can be integrated with other Dash components. The `dash-bio-utils` package additionally converts files from some of the most prominent bioinformatics databases into familiar Python data types such as dictionaries. When used in conjunction with the `dash-bio` package, this enables bioinformaticians to quickly and concisely communicate information among one another, and to the rest of the scientific community.

References

[Mol3D] Autodesk. *Molecule 3D for React*. URL: https://github.com/plotly/molecule-3d-for-react

[PDB] Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., Bourne, P.E.. (2000) *The Protein Data Bank*. Nucleic Acids Research, 28: 235-242. URL:https://www.rcsb.org DOI: 10.1093/nar/28.1.235.

[Man] Bhatnagar, Samir. *manhattanly*. URL: https://github.com/sahirbhatnagar/manhattanly

[cBio] Cerami, E., Gao, J., Dogrusoz, U., Gross, B. E., Sumer, S. O., Aksoy, B. A., Jacobsen, A., Byrne, C. J., Heuer, M. L., Larsson, E., Antipin, Y., Reva, B., Goldberg, A. P., Sander, C., Schultz, N..*The cBio Cancer Genomics Portal: An Open Platform for Exploring Multidimensional Cancer Genomics Data*. Cancer Discov May 1 2012 (2) (5) 401-404; DOI: 10.1158/2159-8290.CD-12-0095.

[BioP] Cock, P. J. A., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F., Wilczynski, B., de Hoon, M. J. L.. *Biopython: freely available Python tools for computational molecular biology and bioinformatics*. Bioinformatics 25 (11), 1422–1423 (2009). DOI: 10.1093/bioinformatics/btp163.

[1bna] PDB ID: 1BNA. Drew, H.R., Wing, R.M., Takano, T., Broka, C., Tanaka, S., Itakura, K., Dickerson, R.E.. *Structure of a B-DNA dodecamer: conformation and dynamics.*. (1981) Proc.Natl.Acad.Sci.USA 78: 2179-2183. DOI: 10.1073/pnas.78.4.2179.

[GEO] Edgar, R., Domrachev, M., Lash, A.E.. *Gene Expression Omnibus: NCBI gene expression and hybridization array data repository*. Nucleic Acids Res. 2002 Jan 1;30(1):207-10. DOI: 10.1093/nar/30.1.207.

[SeqV] FlyBase. *react-sequence-viewer*. URL: https://github.com/FlyBase/react-sequence-viewer

[Ghr] Genome Reference Consortium. *Genome Reference Consortium Human Build 37 (GRCh37)* (2009). URL: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13/

[cBio2] Gao, J., Aksoy, B. A., Dogrusoz, U., Dresdner, G., Gross, B., Sumer, S. O., Sun, Y., Jacobsen, A., Sinha, R. *Integrative Analysis of Complex Cancer Genomics and Clinical Profiles Using the cBioPortal*. Science Signaling Apr 2 2013. DOI: 10.1126/scisignal.2004088.

[Circos] Girault, N.. *circosJS: d3 library to build circular graphs*. URL: https://github.com/nicgirault/circosJS

[GEOP] Gumienny, R.. *GEOparse*. URL: https://github.com/guma44/GEOparse

[JCViz] johnchase. *Visualizing bioinformatics data with plot.ly*. URL:https://plot.ly/~johnchase/22/visualizing-bioinformatics-data-with-plo/

[miR] Kneitz, B., Krebs, M., Kalogirou, C., Schubert, M., et al. *Survival in patients with high-risk prostate cancer is predicted by miR-221, which regulates proliferation, apoptosis, and invasion of prostate cancer cells by inhibiting IRF2 and SOCS3*. Cancer Res 2014 May 1;74(9):2591-603. PMID: 24607843. DOI: 10.1158/0008-5472.CAN-13-1606.

[KR09] Kuner, R., Muley, T., Meister, M., Ruschhaupt, M. et al. *Global gene expression analysis reveals specific patterns of cell junctions in non-small cell lung cancer subtypes.* Lung Cancer 2009 Jan;63(1):32-8. PMID: 18486272. DOI: 10.1016/j.lungcan.2008.03.033.

[NCBI] The NCBI C++ Toolkit (https://ncbi.github.io/cxx-toolkit/) by the National Center for Biotechnology Information, U.S. *Fasta Sequence ID Format*. National Library of Medicine; Bethesda MD, 20894 USA.

[nXP] NeXtprot. *ALB - Serum albumin - proteomics*. URL: https://www.nextprot.org/entry/NX_P02768/proteomics

[Dash] Plotly. *Introducing Dash*. (2017) URL: https://medium.com/@plotlygraphs/introducing-dash-5ecf7191b503

[Align] Plotly. *React Alignment Viewer*. URL: https://github.com/plotly/react-alignment-viewer

[Onco] Plotly. *React OncoPrint*. URL: https://github.com/plotly/react-oncoprint

[Geno] Pritchard, L., White, J. A., Birch, P. R. J., Toth, I. K. *GenomeDiagram: a python package for the visualization of large-scale genomic data*. Bioinformatics, Volume 22, Issue 5, 1 March 2006, Pages 616–617. DOI: 10.1093/bioinformatics/btk021

[ER15] Reed, E., Nunez, S., Kulp, D., Qian, J., Reilly, M. P., and Foulkes, A. S. (2015) *A guide to genome-wide association analysis and post-analytic interrogation.* Statist. Med., 34: 3769– 3792. DOI: 10.1002/sim.6605.

[Muts] Schroeder, M.. *Mutations Needle Plot (muts-needle-plot)*. URL: https://github.com/bbglab/muts-needle-plot

[Par] Swails, J.. *ParmEd*. URL: https://github.com/ParmEd/ParmEd

[Speck] Terrell, R.. *Speck*. URL: https://github.com/wwwtyro/speck

[UniP] The UniProt Consortium. *UniProt: a worldwide hub of protein knowledge*. Nucleic Acids Res. 47: D506-515 (2019). DOI: 10.1093/nar/gky1049.

[Ideo] Weitz, E.. *ideogram: Chromosome visualization with JavaScript*. URL: https://github.com/eweitz/ideogram

[PdbF] wwwPDB. *Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description Version 3.30* (2008). 185-197.

PMDA - Parallel Molecular Dynamics Analysis

Shujie Fan[¶†], Max Linke[‖†], Ioannis Paraskevakos[**], Richard J. Gowers[‡§], Michael Gecht[‖], Oliver Beckstein[¶*]

Abstract—MDAnalysis is an object-oriented Python library to analyze trajectories from molecular dynamics (MD) simulations in many popular formats. With the development of highly optimized MD software packages on high performance computing (HPC) resources, the size of simulation trajectories is growing up to many terabytes in size. However efficient usage of multicore architecture is a challenge for MDAnalysis, which does not yet provide a standard interface for parallel analysis. To address the challenge, we developed PMDA, a Python library that builds upon MDAnalysis to provide parallel analysis algorithms. PMDA parallelizes common analysis algorithms in MDAnalysis through a task-based approach with the Dask library. We implement a simple split-apply-combine scheme for parallel trajectory analysis. The trajectory is split into blocks, analysis is performed separately and in parallel on each block ("apply"), then results from each block are gathered and combined. PMDA allows one to perform parallel trajectory analysis with pre-defined analysis tasks. In addition, it provides a common interface that makes it easy to create user-defined parallel analysis modules. PMDA supports all schedulers in Dask, and one can run analysis in a distributed fashion on HPC machines, ad-hoc clusters, a single multi-core workstation or a laptop. We tested the performance of PMDA on single node and multiple nodes on a national supercomputer. The results show that parallelization improves the performance of trajectory analysis and, depending on the analysis task, can cut down time to solution from hours to minutes. Although still in alpha stage, it is already used on resources ranging from multi-core laptops to XSEDE supercomputers to speed up analysis of molecular dynamics trajectories. PMDA is available as open source under the GNU General Public License, version 2 and can be easily installed via the `pip` and `conda` package managers.

Index Terms—Molecular Dynamics Simulations, High Performance Computing, Python, Dask, MDAnalysis

Introduction

Classical molecular dynamics (MD) simulations have become an invaluable tool to understand the function of biomolecules [KM02], [DDG+12], [SB14], [Oro14], [BLL18], [HBD+19] (often with a view towards drug discovery [BS12]) and diverse problems in materials science [Rot09], [LS15], [VMMC+15], [LJYH18], [KAHC18], [FPM18]. Systems are modeled as particles (for example, atoms) whose interactions are approximated with a classical potential energy function [FS02], [BGM+18]. Forces on the particles are derived from the potential and Newton's equations of motion for the particles are solved with an integrator algorithm, typically using highly optimized MD codes that run on high performance computing (HPC) resources or workstations (often equipped with GPU accelerators). The resulting trajectories, the time series of particle positions $\mathbf{r}(t)$ (and possibly velocities), are analyzed with statistical mechanics approaches [Tuc10], [BGM+18] to obtain predictions or to compare to experimentally measured quantities. Currently simulated systems may contain millions of atoms and the trajectories can consist of hundreds of thousands to millions of individual time frames, thus resulting in file sizes ranging from tens of gigabytes to tens of terabytes. Processing and analyzing these trajectories is increasingly becoming a rate limiting step in computational workflows [CR15], [BFJ18]. Modern MD packages are highly optimized to perform well on current HPC clusters with hundreds of cores such as the XSEDE supercomputers [TCD+14] but current general purpose trajectory analysis packages [Gio19] were not designed with HPC in mind.

In order to scale up trajectory analysis from workstations to HPC clusters with the MDAnalysis Python library [MADWB11], [GLB+16] we leveraged Dask [Roc15], [Das16], a task-graph parallel framework, together with Dask's various schedulers (in particular *distributed*), and created the *Parallel MDAnalysis* (PMDA) library. By default, PMDA follows a simple split-apply-combine [Wic11] approach for trajectory analysis, whereby each task analyzes a single trajectory segment and reports back the individual results that are then combined into the final result [KPJB17]. Our previous work established that Dask worked well with MDAnalysis [KPJB17] and that this approach was competitive with other task-parallel approaches [PLK+18]. However, we did not provide a general purpose framework to write parallel analysis tools with MDAnalysis. Here we show how the split-apply-combine approach lends itself to a generalizable Python implementation that makes it straightforward for users to implement their own parallel analysis tools. At the heart of PMDA is the idea that the user only needs to provide a function that analyzes a single trajectory frame. PMDA provides the remaining framework via the `ParallelAnalysisBase` class to split the trajectory, apply the user's function to trajectory frames, run the analysis in parallel via Dask/*distributed*, and combines the data. It also contains a growing library of ready-to-use analysis classes, thus enabling users to immediately accelerate analysis that they previously performed in serial with the standard MDAnalysis analysis classes [GLB+16].

Methods

At the core of PMDA is the idea that a common interface makes it easy to create code that can be easily parallelized, especially

† *These authors contributed equally.*
¶ *Arizona State University*
‖ *Max Planck Institute of Biophysics*
** *Rutgers University*
‡ *University of New Hampshire*
§ *present address: NextMove Software Ltd.*
* *Corresponding author: obeckste@asu.edu*

if the analysis can be split into independent work over multiple trajectory slices and a final step, in which all data from the trajectory slices are combined. We first describe typical steps in analyzing MD trajectories and then outline the approach taken in PMDA.

Trajectory analysis

A trajectory with T saved time steps consists of a sequence of coordinates $\{(\mathbf{r}_1(t), \mathbf{r}_2(t), \ldots \mathbf{r}_N(t))\}_{1 \le t \le T}$ where $\mathbf{r}_i(t)$ are the Cartesian coordinates of particle i at time step t with N particles in the simulated system, i.e., $T \times N \times 3$ floating point numbers in total. To simplify notation, we consider t as an integer that indexes the trajectory frames; each frame index corresponds to a physical time in the trajectory that we could obtain if needed. In general, the coordinates are passed to a function $\mathscr{A}(\{\mathbf{r}_i(t)\})$ to compute a time-dependent quantity

$$A(t) = \mathscr{A}(\{\mathbf{r}_i(t)\}). \tag{1}$$

This quantity does not have to be a simple scalar; it may be a vector or a function of another parameter. In many cases, the *time series* $A(t)$ is the desired result. It is, however, also common to perform some form of *reduction* on the data, which can be as simple as a time average to compute a thermodynamic average $\langle A \rangle \equiv \bar{A} = T^{-1} \sum_{t=1}^{T} A(t)$. Such an average can be easily calculated in a post-analysis step after the time series has been obtained. An example of a more complicated reduction is the calculation of a histogram such as a radial distribution function (RDF) [FS02], [Tuc10] between two types of particles with numbers N_a and N_b,

$$g(r) = \left\langle \frac{1}{N_a N_b} \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} \delta(|\mathbf{r}_{a,i} - \mathbf{r}_{b,j}| - r) \right\rangle \tag{2}$$

where the Dirac delta function counts the occurrences of particles i and j at distance r. To compute a RDF, we could generate a time series of histograms along the spatial coordinate r, i.e., $A(t;r)$ for each frame, and then perform the average in post-analysis. However, storage of such histograms becomes problematic, especially if instead of 1-dimensional RDFs, densities on 3-dimensional grids are being calculated. It is therefore better to reformulate the algorithm to perform a partial average (or reduction) during the analysis on a per-frame basis. For histograms, this could mean building a partial histogram and updating counts in the bins after every frame. PMDA supports the simple time series data collection and the per-frame reduction.

Split-apply-combine

The *split-apply-combine* strategy can be thought of as a simplified map-reduce [Wic11] that provides a conceptually simple approach to operate on data in parallel. It is based on the fundamental assumption that the data can be partitioned into blocks that can be analyzed independently. The trajectory is split along the time axis into M blocks of approximately equal size, $\tau = T/M$. One trajectory block can be viewed as a slice of a trajectory, e.g., for block k, $\{(\mathbf{r}_1(t), \mathbf{r}_2(t), \ldots \mathbf{r}_N(t))\}_{t_k \le t < t_k + \tau_k}$ with τ_k frames in the block. Each block k is analyzed in parallel by applying the function $\mathscr{A}$ to the frames in each block. Finally, the results from all blocks are gathered and combined.

The advantage of this approach is its simplicity. Many typical analysis tasks are based on calculations of time series from single trajectory frames as in Eq. 1 and it is this calculation that varies from task to task while the book-keeping and trajectory slicing is the same. Given a function $\mathscr{A}$ that performs the *single frame calculation*, PMDA provides code to perform the other necessary steps (Fig. 1).

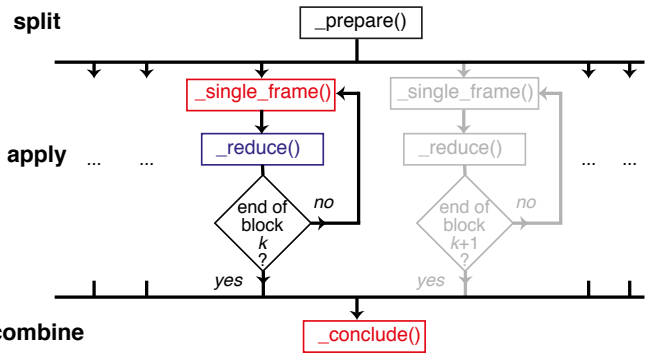


Fig. 1: *High-level view of the split-apply-combine algorithm in PMDA. Steps are labeled with the methods in* `pmda.parallel.ParallelAnalysisBase` *that perform the corresponding function. Methods in red (*`_single_frame()` *and* `_conclude()`*) must be implemented for every analysis function because they are not general. The blue method* `_reduce()` *must be implemented unless a simple time series is being calculated. The* `_prepare()` *method is optional and provides a hook to initialize custom data structures.*

As explained in more detail later, a class derived from `pmda.parallel.ParallelAnalysisBase` encapsulates one trajectory analysis calculation. Individual methods correspond to different steps and in the following (and in Fig. 1) we will mention the names of the relevant methods to make clear how PMDA abstracts parallel analysis. The calculation with M parallel workers is *prepared* by setting up data structures to hold the final result (method `_prepare()`). The indices for the M trajectory slices are created in such a way that the number of frames τ_k are balanced and do not differ by more than 1. For each slice or block k, the *single frame* analysis function $\mathscr{A}$ (`_single_frame()`) is sequentially applied to all frames in the slice. The result, $A(t)$, is *reduced*, i.e., added to the results for this block. For time series, $A(t)$ is simply appended to a list to form a partial time series for the block. More complicated reductions (method `_reduce()`) can be implemented, for example, the data may be histogrammed and added to a partial histogram for the block (as necessary for the implementation of the parallel RDF Eq. 2).

Implementation

PMDA is written in Python and, through MDAnalysis [GLB+16], reads trajectory data from the file system into NumPy arrays [Oli07], [VDWCV11]. Dask's `delayed()` function is used to build a task graph that is then executed using any of the schedulers available to Dask [Das16].

MDAnalysis combines a trajectory file (frames of coordinates that change with time) and a topology file (list of particles, their names, charges, bonds — all information that does not change with time) into a `Universe(topology, trajectory)` object. Arbitrary selections of particles (often atoms) are made available as an `AtomGroup` and the common approach in MDAnalysis is to work with these objects [GLB+16]; for instance, all coordinates of an `AtomGroup` with N atoms named `protein` are accessed as the $N \times 3$ NumPy array `protein.positions`.

`pmda.parallel.ParallelAnalysisBase` is the base class for defining a split-apply-combine parallel multi frame analysis in PMDA. It requires a `Universe` to operate on and any `AtomGroup` instances that will be used. A parallel analysis class must be derived from `ParallelAnalysisBase` and at a minimum, must implement the `_single_frame(ts, agroups)` and `_conclude()` methods. The arguments of `_single_frame(ts, agroups)` are a MDAnalysis `Timestep` instance and a tuple of `AtomGroup` instances so that the following code could be run (the code is a simplified version of the current implementation):

```
1 @delayed
2 def analyze_block(blockslice):
3     result = []
4     for ts in u.trajectory[blockslice]:
5         A = self._single_frame(ts, agroups)
6         result.append(A)
7     return result
```

The task graph is constructed by wrapping the above code into `delayed()` and appending a delayed instance for each trajectory slice to a (delayed) list:

```
7 blocks = delayed([analyze_block(blockslice)
8                   for blockslice in slices])
9 results = blocks.compute(**scheduler_kwargs)
```

Calling the `compute()` method of the delayed list object hands the task graph over to the scheduler, which then executes the graph on the available Dask workers. For example, the *multiprocessing* scheduler can be used to parallelize task graph execution on a single multiprocessor machine while the *distributed* scheduler is used to run on multiple nodes of a HPC cluster. After all workers have finished, the variable `results` contains a list of results from the individual blocks. PMDA actually stores these raw results as `ParallelAnalysisBase._results` and leaves it to the `_conclude()` method to process the results; this can be as simple as `numpy.hstack(self._results)` to generate a time series by concatenating the individual time series from each block.

The default `_reduce()` method appends the results and is equivalent to line 6. In general, line 6 reads

```
6         result = self._reduce(result, A)
```

where variable `result` should have been properly initialized in `_prepare()`. In order to be parallelizable, the `_reduce()` method must be a static method that does not access any class variables but returns its modified first argument. For example, the default "append" reduction is

```
@staticmethod
def _reduce(res, result_single_frame):
    res.append(result_single_frame)
    return res
```

In general, the `ParallelAnalysisBase` controls access to instance attributes via a context manager `ParallelAnalysisBase.readonly_attributes()`. It sets them to "read-only" for all parallel parts to prevent the common mistake to set an instance attribute in a parallel task, which breaks under parallelization as the value of an attribute in an instance in a parallel process is never communicated back to the calling process.

Using PMDA

PMDA allows one to perform parallel trajectory analysis with pre-defined analysis tasks. In addition, it provides a common interface that makes it easy to create user-defined parallel analysis modules. Here, we will introduce some basic usages of PMDA.

Pre-defined Analysis

PMDA contains a growing number of pre-defined analysis classes that are modeled after functionality in `MDAnalysis.analysis` and that can be used right away. Current examples are `pmda.rms` for RMSD analysis, `pmda.contacts` for native contacts analysis, `pmda.rdf` for radial distribution functions, and `pmda.leaflet` for the LeafletFinder analysis tool [MADWB11], [PLK+18] for the topological analysis of lipid membranes. While the first three modules are based on `pmda.parallel.ParallelAnalysisBase` as described above and follow the strict split-apply-combine approach, `pmda.leaflet` is an example of a more complicated task-based algorithm that can also easily be implemented with MDAnalysis and Dask [PLK+18]. All PMDA classes can be used in a similar manner to classes in `MDAnalysis.analysis`, which makes it easy for users of MDAnalysis to switch to parallelized versions of the algorithms. One example is the calculation of the root mean square distance (RMSD) of C_α atoms of the protein with `pmda.rms.RMSD`. An analysis class object is instantiated with the necessary input data such as the `AtomGroup` containing the C_α atoms and a reference structure. To perform the analysis, the `run()` method is called.

```
import MDAnalysis as mda
from pmda import rms
# Create a Universe based on simulation topology
# and trajectory
u = mda.Universe(top, trj)

# Select all the C alpha atoms
ca = u.select_atoms('name CA')

# Take the initial frame as the reference
u.trajectory[0]
ref = u.select_atoms('name CA')

# Build the parallel rms object, and run
# the analysis with 4 workers and 4 blocks.
rmsd = rms.RMSD(ca, ref)
rmsd.run(n_jobs=4, n_blocks=4)

# The results can be accessed in rmsd.rmsd.
print(rmsd.rmsd)
```

Here the only difference between using the serial version and the parallel version is that the `run()` method takes additional arguments `n_jobs` and `n_blocks`, which determine the level of parallelization. When using the *multiprocessing* scheduler (the default), `n_jobs` is the number of processes to start and typically the number of blocks `n_blocks` is set to the number of available CPU cores. When the *distributed* scheduler is used, Dask will automatically learn the number of available Dask worker processes and `n_jobs` is meaningless; instead it makes more sense to set the number of trajectory blocks that are then spread across all available workers.

User-defined Analysis

PMDA makes it easy to create analysis classes such as the ones discussed above. If the per-frame analysis can be expressed as a simple function, then an analysis class can be created with a factory function. Otherwise, a class has to be derived from `pmda.parallel.ParallelAnalysisBase`. Both approaches are described below.

`pmda.custom.AnalysisFromFunction()`: PMDA provides helper functions in `pmda.custom` to rapidly build a parallel class for users who already have a *single frame* function that 1. takes one or more `AtomGroup` instances as input, 2. analyzes one frame in a trajectory and returns the result for this frame. For example, if we already have a function to calculate the radius of gyration [MM14] of a protein given in `AtomGroup ag`, namely `ag.radius_of_gyration()` (as available in MDAnalysis), then we can write a simple function `rgyr()` that returns for each trajectory frame a tuple containing the time at the current time step and the value of the radius of gyration:

```
import MDAnalysis as mda
u = mda.Universe(top, traj)
protein = u.select_atoms('protein')

def rgyr(ag):
    return (ag.universe.trajectory.time,
            ag.radius_of_gyration())
```

We can wrap `rgyr()` in the `pmda.custom.AnalysisFromFunction()` class instance factory function to build a parallel version of `rgyr()`:

```
import pmda.custom
parallel_rgyr = pmda.custom.AnalysisFromFunction(
                rgyr, u, protein)
```

This new parallel analysis class can be run just as the existing ones:

```
parallel_rgyr.run(n_jobs=4, n_blocks=4)
print(parallel_rgyr.results)
```

The time series of the results is stored in the attribute `parallel_rgyr.results`; for our example where each per-frame result is a tuple `(time, Rgyr)`, the time series is stored as a $T \times 2$ array that can be plotted with

```
import matplotlib.pyplot as plt
data = parallel_rgyr.results
plt.plot(data[:, 0] , data[:, 1])
```

`pmda.parallel.ParallelAnalysisBase`: For more general cases, one can write the parallel class with the help of `pmda.parallel.ParallelAnalysisBase`, following the schema in Fig. 1. To build a new analysis class, one should derive a class from `pmda.parallel.ParallelAnalysisBase` that implements

1) the single frame analysis method `_single_frame()` (*required*),
2) the final results conclusion method `_conclude()` (*required*),
3) the additional preparation method `_prepare()` (*optional*),
4) the reduce method for frames within the same block `_reduce()` (*optional* for time series, *required* for anything else).

As an example, we show how one can build a class to calculate the radius of gyration of a protein given in `AtomGroup protein`; of course, in this case the simple approach with `pmda.custom.AnalysisFromFunction()` would be easier.

```
import numpy as np
from pmda.parallel import ParallelAnalysisBase

class RGYR(ParallelAnalysisBase):
    def __init__(self, protein):
        universe = protein.universe
        super(RGYR, self).__init__(universe,
                                   (protein,))
    def _prepare(self):
        self.rgyr = None
    def _conclude(self):
        self.rgyr = np.vstack(self._results)
```

The `_conclude()` method reshapes the attribute `self._results`, which always holds the results from all blocks, into a time series. The call signature for method `_single_frame()` is fixed and `ts` must contain the current MDAnalysis `Timestep` and `agroups` must be a tuple of `AtomGroup` instances. The current frame number, time and radius of gyration are returned as the single frame results:

```
def _single_frame(self, ts, atomgroups):
    protein = atomgroups[0]
    return (ts.frame, ts.time,
            protein.radius_of_gyration())
```

Because we want to return a time series, it is not necessary to define a `_reduce()` method. This class can be used in the same way as the class that we defined with `pmda.custom.AnalysisFromFunction`:

```
parallel_rgyr = RGYR(protein)
parallel_rgyr.run(n_jobs=4, n_blocks=4)
print(parallel_rgyr.results)
```

Performance Evaluation

In order to characterize the performance of PMDA on a typical HPC machine we performed computational experiments for two different analysis tasks, the RMSD calculation after optimum superposition (*RMSD*) and the water oxygen radial distribution function (*RDF*).

For the *RMSD* task we computed the time series of root mean square distance after optimum superposition (RMSD) of all 564 C_α atoms of a protein with the initial coordinates at the first frame as reference, as implemented in class `pmda.rms.RMSD`. The RMSD calculation with optimum superposition was performed with the fast QCPROT algorithm [The05] as implemented in MDAnalysis [MADWB11].

As a second test case we computed the water oxygen-oxygen radial distribution function (*RDF*, Eq. 2) in 75 bins up to a cut-off of 5 Å for all 24,239 oxygen atoms in the water molecules in our test system, using the class `pmda.rdf.InterRDF`. The RDF calculation is compute-intensive due to the necessity to calculate and histogram a large number ($\mathcal{O}(N)$ because of the use of a cut-off) of distances for each time step; it additionally exemplifies a non-trivial reduction.

These two common computational tasks differ in their computational cost and represent two different requirements for data reduction and thus allow us to investigate two distinct use cases. We investigated a long (9000 frames) and a short trajectory (900 frames) to assess to which degree parallelization remained practical. The computational experiments were performed in different scenarios to assess the influence of different Dask schedulers (*multiprocessing* and *distributed*) and the role of the file storage system (shared Lustre parallel file system and local SSD), as described below and summarized in Table 1.

Test system, benchmarking environment, and data files

We tested PMDA 0.2.1, MDAnalysis 0.20.0 (development version), Dask 1.2.0, and NumPy 1.15.4 under Python 3.6. All

configuration label	file storage	scheduler	max nodes	max processes
Lustre-distributed-3nodes	Lustre	*distributed*	3	72
Lustre-distributed-6nodes	Lustre	*distributed*	6	72
Lustre-multiprocessing	Lustre	*multiprocessing*	1	24
SSD-distributed	SSD	*distributed*	1	24
SSD-multiprocessing	SSD	*multiprocessing*	1	24

TABLE 1: *Testing configurations on SDSC Comet.* ***max nodes*** *is the maximum number of nodes that were tested; the multiprocessing scheduler is limited to a single node.* ***max processes*** *is the maximum number of processes or Dask workers that were employed.*

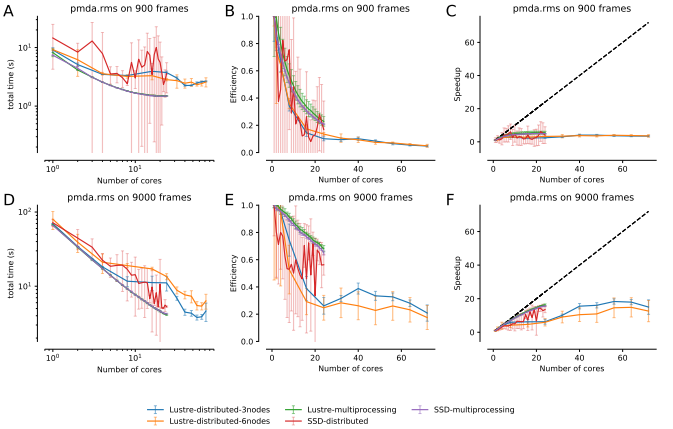


Fig. 2: *Strong scaling performance of the RMSD analysis task with short (900 frames) and long (9000) frames trajectories on* SDSC Comet*, where a single node contains 24 cores. The total time to completion t^{total} was measured for different testing configurations (Table 1).* ***A*** *and* ***D****: t^{total} as a function of processes or Dask workers, i.e., the number of CPU cores that were actually used. The number of trajectory blocks was the same as the number of CPU cores.* ***B*** *and* ***E****: efficiency E. The ideal case is $E = 1$.* ***C*** *and* ***F****: speed-up S. The dashed line represents ideal strong scaling $S(M) = M$. Points represent the mean over five repeats with the standard deviation shown as error bars.*

packages except PMDA and MDAnalysis were installed with the conda package manager from the conda-forge channel. PMDA and MDAnalysis development versions were installed from source in a conda environment with `pip install`.

Benchmarks were run on the CPU nodes of XSEDE's [TCD+14] *SDSC Comet* supercomputer, a 2 PFlop/s cluster with 1,944 Intel Haswell Standard Compute Nodes in total. Each node contains two Intel Xeon CPUs (E5-2680v3, 12 cores, 2.5 GHz) with 24 CPU cores per node, 128 GB DDR4 DRAM main memory, and a non-blocking fat-tree InfiniBand FDR 56 Gbps node interconnect. All nodes share a Lustre parallel file system and have access to node-local 320 GB SSD scratch space. Jobs are run through the SLURM batch queuing system. Our SLURM submission shell scripts and Python benchmark scripts for *SDSC Comet* are available in the repository https://github.com/Becksteinlab/scipy2019-pmda-data and are archived under DOI 10.5281/zenodo.3228422.

The test data files consist of a topology file `YiiP_system.pdb` (with $N = 111{,}815$ atoms) and two trajectory files `YiiP_system_9ns_center.xtc` (Gromacs XTC format, $T = 900$ frames) and `YiiP_system_90ns_center.xtc` (Gromacs XTC format, $T = 9000$ frames) of the membrane protein YiiP in a lipid bilayer together with water and ions. The test trajectories are made available on figshare at DOI 10.6084/m9.figshare.8202149.

We tested different combinations of Dask schedulers (*distributed*, *multiprocessing*) with different means to read the trajectory data (either from the shared Lustre parallel file system or from local SSD) as shown in Table 1. Using either the *multiprocessing* scheduler or the SSD restrict runs to a single node (maximum 24 CPU cores). With *distributed* (and Lustre) we tested fully utilizing all cores on a node and also only occupying half the available cores, while doubling the total number of nodes. In all cases the trajectory were split in as many blocks as there were available processes or Dask workers. We performed five independent repeat runs for all scenarios in Table 1 and plotted the mean of the reported timing quantity together with the standard deviation from the mean to indicate the variance of the runs.

Measured timing quantities

The `ParallelAnalysisBase` class collects detailed timing information for all blocks and all frames and makes these data available in the attribute `ParallelAnalysisBase.timing`: We measured the time t^{prepare} for `_prepare()`, the time t_k^{wait} that each task k waits until it is executed by the scheduler, the time t_k^{Universe} to create a new `Universe` for each Dask task (which includes opening the shared trajectory and topology files and loading the topology into memory), the time $t_{k,t}^{\text{I/O}}$ to read each frame t in each block k from disk into memory, the time $t_{k,t}^{\text{compute}}$ to perform the computation in `_single_frame()` and reduction in `_reduce()`, the time t_k^{conclude} to perform the final processing of all data in `_conclude()`, and the total wall time to solution t^{total}.

We analyzed the total time to completion as a function of the number of CPU cores, which was equal to the number of trajectory blocks, so that each block could be processed in parallel. We quantified the strong scaling behavior by calculating the *speed-up* for running on M CPU cores with M parallel Dask tasks as $S(M) = t^{\text{total}}(1)/t^{\text{total}}(M)$, where $t^{\text{total}}(1)$ is the performance of the PMDA code using the serial scheduler. The *efficiency* was calculated as $E(M) = S(M)/M$. The errors of these quantities were derived by the standard error propagation.

To gain better insight into the performance-limiting steps in our algorithm (Fig. 1) we plotted the *maximum* times over all ranks because the overall time to completion cannot be faster than the slowest parallel process. For example, for the read I/O time we calculated the total read I/O time for each rank k as $t_k^{\text{I/O}} = \sum_{t=t_k}^{t_k+\tau_k} t_{k,t}^{\text{I/O}}$ and then reported $\max_k t_k^{\text{I/O}}$.

RMSD analysis task

The parallelized RMSD analysis in `pmda.rms.RMSD` scaled well only to about half a node (12 cores), as shown in Fig. 2 A, D, regardless of the length of the trajectory. The efficiency dropped below 0.8 (Fig. 2 B, E) and the maximum achievable speed-up remained below 10 for the short trajectory (Fig. 2 C) and below 20 for the long one (Fig. 2 F). Overall, using the *multiprocessing* scheduler and either Lustre or SSD gave the best performance and shortest time to solution. The *distributed* scheduler with SSD gave widely variable results as seen by large standard deviations over

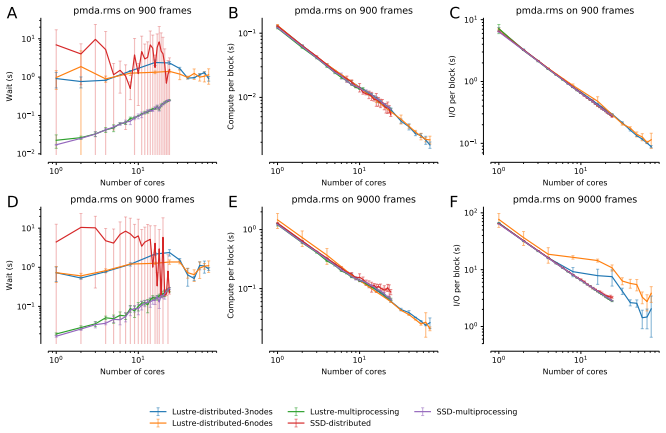


Fig. 3: *Detailed per-task timing analysis for parallel components of RMSD analysis task. Individual times per task were measured for different testing configurations (Table 1). **A** and **D**: Maximum waiting time for the task to be executed by the Dask scheduler. **B** and **E**: Maximum total compute time per task. **C** and **F**: Maximum total read I/O time per task. Points represent the mean over five repeats with the standard deviation shown as error bars.*

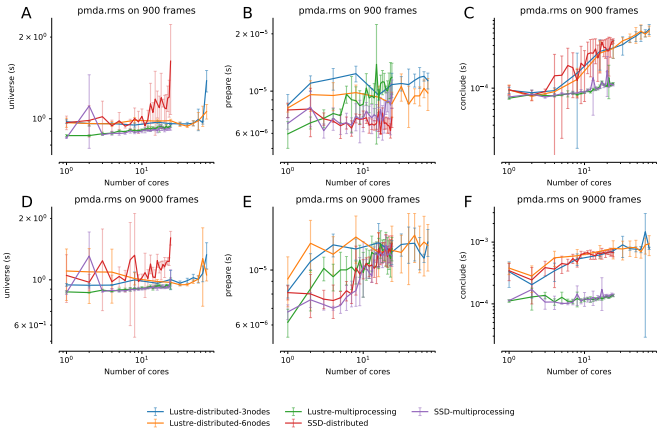


Fig. 4: *Detailed timing analysis for other components of the RMSD analysis task. Individual times per task were measured for different testing configurations (Table 1). **A** and **D**: Maximum time for a task to load the* `Universe`*. **B** and **E**: Time $t^{prepare}$ to execute* `_prepare()`*. **C** and **F**: Time $t_k^{conclude}$ to execute* `_conclude()`*. Points represent the mean over five repeats with the standard deviation shown as error bars.*

multiple repeats. It still performed better than when the Lustre file system was used but overall, for a single node, the *multiprocessing* scheduler always gave better performance with less variation in run time. These results were consistent with findings in our earlier pilot study where we had looked at the RMSD task with Dask and had found that *multiprocessing* with both SSD and Lustre had given good single node performance but, using *distributed*, had not scaled well beyond a single *SDSC Comet* node [KPJB17].

A detailed look at the maximum times (Fig. 3) that the Dask worker processes spent on waiting to be executed, performing the RMSD calculation with data in memory, and reading the trajectory frame data from the file into memory showed that the waiting time (Fig. 3 A, D) either increased from about 0.02 s to 0.1 s for *multiprocessing* or was roughly a constant 1 s for *distributed* (on Lustre). For reasons that were not clear, the *distributed* scheduler with SSD had on average the largest wait times, with large fluctuations, ranging from 0.1 s to 10 s (red lines in Fig. 3 A, D). The computation itself scaled very well (Fig. 3 B, E) with only small variations, indicating that split-apply-combine is a robust approach to parallelization, once the data are in memory. The reading time scaled fairly well but exhibited some variation beyond a single node (24 cores) and an unexplained decline in performance for the longer trajectory, as seen in Fig. 3 C, F. The read I/O results indicated that both Lustre and SSD can perform equally well. Beyond 12 cores, the waiting time started approaching the time for read I/O (compute was an order of magnitude less than I/O) and hence parallel speed-up was limited by the wait time.

The second major component that limited scaling performance was the time to create the `Universe` data structure (Fig. 4 A, D). The time to read the topology and open the trajectory file on the shared file system typically increased from 1 s to about 2 s and thus, for the given total trajectory lengths, also became comparable to the time for read I/O. The other components (prepare and conclude, see Fig. 4) remained negligible with times below 10^{-3} s.

The parallelizable fraction of the workload consisted of the

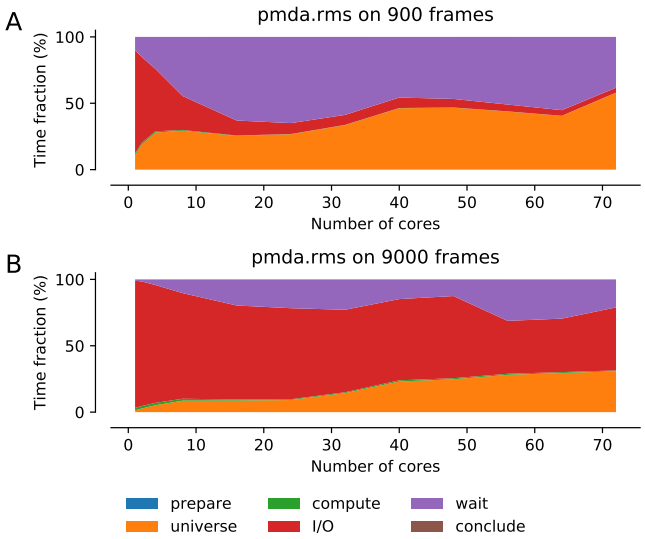


Fig. 5: *Fraction of the total run time taken by individual steps in the parallel* RMSD *calculation for* distributed *on up to three nodes (Lustre-distributed-3nodes). Compute (green) and read I/O (red) represent the parallelizable fraction of the program; all other components are effectively serial. **A** Trajectory with 900 frames. **B** Trajectory with 9000 frames.*

compute and read I/O steps. Because this fraction was relatively small and was dominated by the wait time from the Dask scheduler and the time to initialize the `Universe` data structure (Fig. 5), the overall performance gain by parallelization remained modest, as explained by Amdahl's law [Amd67]. Thus, for a highly optimized and fast computation such as the RMSD calculation, the best performance (speed-up on the order of 10 fold) could already be achieved on the equivalent of a modern workstation. The *multiprocessing* scheduler seemed to be the more consistent and better performing choice in this scenario; therefore PMDA defaults to *multiprocessing*. Performance would likely improve with longer trajectories because the "fixed" serial costs (waiting, `Universe`

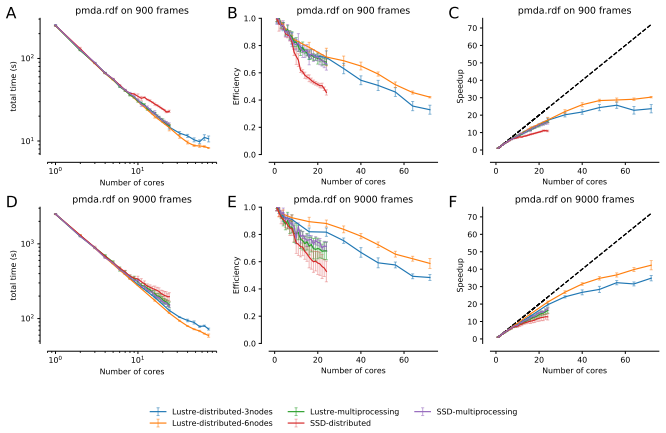


Fig. 6: *Strong scaling performance of the RDF analysis task. The total time to completion t^{total} was measured for different testing configurations (Table 1). **A** and **D**: t^{total} as a function of processes or Dask workers, i.e., the number of CPU cores that were actually used. The number of trajectory blocks was the same as the number of CPU cores. **B** and **E**: efficiency E. The ideal case is $E = 1$. **C** and **F**: speed-up S. The dashed line represents ideal strong scaling $S(M) = M$. Points represent the mean over five repeats with the standard deviation shown as error bars.*

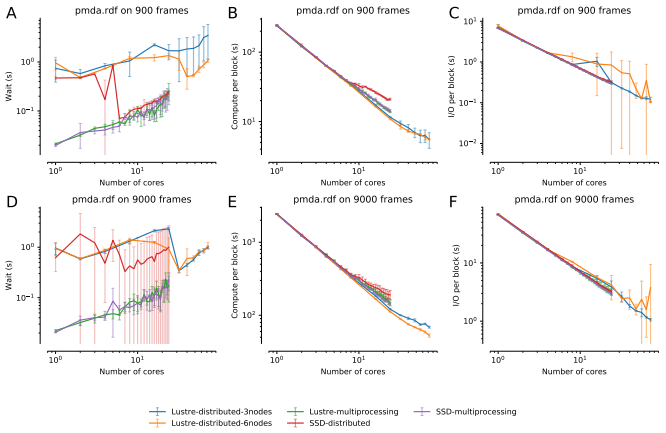


Fig. 7: *Detailed per-task timing analysis for parallel components of the RDF analysis task. Individual times per task were measured for different testing configurations (Table 1). **A** and **D**: Maximum waiting time for the task to be executed by the Dask scheduler. **B** and **E**: Maximum total compute time per task. **C** and **F**: Maximum total read I/O time per task. Points represent the mean over five repeats with the standard deviation shown as error bars.*

creation) would decrease in relevance to the time spent on computation and data ingestion (see Fig. 5 B), which benefit from parallelization [Gus88]. However, all things considered, a single node seemed sufficient to accelerate RMSD analysis.

RDF analysis task

Unlike the RMSD analysis task, the parallelized RDF analysis in `pmda.rdf.InterRDF` showed decreasing total time to solution up to the highest number of CPU cores tested (see Fig. 6 A, D). The efficiency on a single node remained above 0.6 for almost all cases (Fig. 6 B, E) and remained above 0.6 for the best case (*distributed* on Lustre and half-filling of nodes for the long trajectory), up to 3 nodes (72 cores, Fig. 6 E). Even when filling complete nodes, the efficiency for the long trajectory remained above 0.5 (Fig. 6 E). Consequently, a sizable speed-up could be maintained that approached 40 fold in the best case (Fig. 6 F), which cut down the time to solution from about 40 min to under 1 min. On a single node, all approaches performed similarly well, with the *distributed* scheduler now having a slight edge over *multiprocessing* (Fig. 6), with the exception of the combination of *distributed* with the SSD, which for unknown reasons performed much worse than everything else (similar to the situation observed for the *RMSD* case).

The detailed analysis of the individual components in Fig. 7 clearly showed that the RDF analysis task required much more computational effort than the RMSD task and that it was dominated by the compute component (Fig. 8), which scaled very well to the highest core numbers (Fig. 7 B, E). However, *multiprocessing* and especially *distributed* with *SSD* took longer for the computational part at ≥ 8 cores (one third of a single node), indicating that in these cases some sort of competition reduced performance. For comparison, serial computation required about 250 s while read I/O required less than 10 s, and this ratio was approximately maintained as the read I/O also scaled reasonably well (Fig. 7 C, F) Although the variance increased markedly when multiple nodes were included such as when using six half-filled nodes, this effect did not strongly impact overall performance because $t_{k,t}^{\text{compute}} \gg t_{k,t}^{\text{I/O}}$. The differences between using all cores on a node compared to only using half the cores on each node were small but only using half a node was consistently better, especially in the compute time, and hence the overall performance of the latter approach was better. For the shorter trajectory, the wait time was a factor in reducing performance at higher core numbers (Fig. 7 A). The other components ($t_k^{\text{Universe}} < 2$ s, $t^{\text{prepare}} < 3 \times 10^{-5}$ s , $t_k^{\text{conclude}} < 4 \times 10^{-4}$ s) were similar or better (i.e., shorter) than the ones shown for the RMSD task in Fig. 4 and are not shown; only the time to set up the `Universe` played a role in reducing the scaling performance in the *Lustre-distributed-3nodes* scenario at 60 or more CPU cores.

In summary, the performance increase for a compute-intensive task such as RDF was sizable and, although not extremely efficient, was large enough (about 30-40) to justify the use of about 100 cores on a HPC supercomputer. Because scaling seemed mostly limited by constant costs such as the scheduling wait time (see Fig. 8), processing longer trajectories, for which more work has to be done in the parallelizable compute and read I/O steps, should improve the scaling behavior [Gus88].

Conclusions

The PMDA Python package provides a framework to parallelize analysis of MD trajectories with a simple *split-apply-combine* approach by combining Dask with MDAnalysis. Although still in early development, it provides useful functionality for users to speed up analysis, ranging from a growing library of included tools to different approaches for users to write their own parallel analysis. In simple cases, just wrapping a user supplied function is enough to immediately use PMDA but the package also provides a documented API to derive from the `pmda.parallel.ParallelAnalysisBase` class. We showed that performance depends on the type of analysis that is being performed. Compute-intensive tasks such as the RDF calculation can show good strong scaling up to about a hundred cores on a typical supercomputer and speeding up the time to

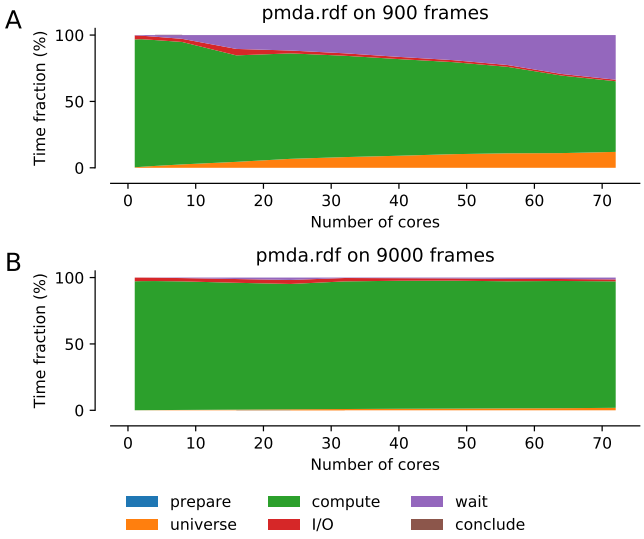


Fig. 8: *Fraction of the total run time taken by individual steps in the parallel* RDF *calculation for* distributed *on up to three nodes (Lustre-distributed-3nodes). Compute (green) and read I/O (red) represent the parallelizable fraction of the program; all other components are effectively serial.* ***A*** *Trajectory with 900 frames.* ***B*** *Trajectory with 9000 frames.*

solution from hours in serial to minutes in parallel should make this an attractive solution for many users. For other analysis tasks such as the RMSD calculation and other similar ones (e.g., simple distance calculations), a single multi-core workstation seems sufficient to achieve speed-ups on the order of 10 and HPC resources would not be useful. But thanks to the design of Dask, running a PMDA analysis on a laptop, workstation, or supercomputer requires absolutely no changes in the code and users are free to immediately choose the computational resource that best fits their purpose.

Code availability and development process

PMDA is available in source form under the GNU General Public License v2 from the GitHub repository MDAnalysis/pmda, and as a PyPi package and conda package (via the conda-forge channel). Python 2.7 and Python $\geq$ 3.5 are fully supported and tested. The package uses semantic versioning to make it easy for users to judge the impact of upgrading. The development process uses continuous integration (Travis CI): extensive tests are run on all commits and pull requests via pytest, resulting in a current code coverage of 97% and documentation is automatically generated by Sphinx and published as GitHub pages. Users are supported through the community mailing list (Google group) and the GitHub issue tracker.

Acknowledgments

We would like to thank reviewer Cyrus Harrison for the idea to plot the fractional time spent on different stages of the program (Figs. 5 and 8). This work was supported by the National Science Foundation under grant numbers ACI-1443054 and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. The SDSC Comet computer at the San Diego Supercomputer Center was used under allocation TG-MCB130177. Max Linke was supported by NumFOCUS under a small development grant.

References

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi:10.1145/1465482.1465560.

[BFJ18] Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Convergence of data generation and analysis in the biomolecular simulation community. In *Online Resource for Big Data and Extreme-Scale Computing Workshop*, November 2018. URL: https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/Beckstein-Fox-Jha_BDEC2_WP_0.pdf.

[BGM+18] Efrem Braun, Justin Gilmer, Heather B. Mayes, David L. Mobley, Jacob I. Monroe, Samarjeet Prasad, and Daniel M. Zuckerman. Best practices for foundations in molecular simulations [article v1.0]. *Living Journal of Computational Molecular Science*, 1(1):5957–, 11 2018. doi:10.33011/livecoms.1.1.5957.

[BLL18] Sandro Bottaro and Kresten Lindorff-Larsen. Biophysical experiments and biomolecular simulations: A perfect match? *Science*, 361(6400):355–360, 2018. doi:10.1126/science.aat4010.

[BS12] David W Borhani and David E Shaw. The future of molecular dynamics simulations in drug discovery. *J Comput Aided Mol Des*, 26(1):15–26, Jan 2012. doi:10.1007/s10822-011-9517-y.

[CR15] T. Cheatham and D. Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering*, 17(2):30–39, 2015. doi:10.1109/MCSE.2015.7.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[DDG+12] Ron O Dror, Robert M Dirks, J P Grossman, Huafeng Xu, and David E Shaw. Biomolecular simulation: a computational microscope for molecular biology. *Annu Rev Biophys*, 41:429–52, 2012. doi:10.1146/annurev-biophys-042910-155245.

[FPM18] Pim W J M Frederix, Ilias Patmanidis, and Siewert J Marrink. Molecular simulations of self-assembling bio-inspired supramolecular systems and their connection to experiments. *Chem Soc Rev*, 47(10):3470–3489, May 2018. doi:10.1039/c8cs00040a.

[FS02] Daan Frenkel and Berend Smit. *Understanding Molecular Simulations*. Academic Press, San Diego, 2 edition, 2002.

[Gio19] Toni Giorgino. Analysis libraries for molecular trajectories: a cross-language synopsis. In M. Bonomi and C. Camilloni, editors, *Biomolecular Simulations: Methods and Protocols*. Springer, 2019.

[GLB+16] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, David L Dotson, Jan Domański, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98–105, Austin, TX, 2016. SciPy. URL: https://www.mdanalysis.org, doi:10.25080/Majora-629e541a-00e.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. doi:10.1145/42411.42415.

[HBD+19] David J. Huggins, Philip C. Biggin, Marc A. Dämgen, Jonathan W. Essex, Sarah A. Harris, Richard H. Henchman, Syma Khalid, Antonija Kuzmanic, Charles A. Laughton, Julien Michel, Adrian J. Mulholland, Edina Rosta, Mark S. P. Sansom, and Marc W. van der Kamp. Biomolecular simulations: From dynamics and mechanisms to computational assays of biological activity. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 9(3):e1393, 2019. doi:10.1002/wcms.1393.

[KAHC18] Grit Kupgan, Lauren J. Abbott, Kyle E. Hart, and Coray M. Colina. Modeling amorphous microporous polymers for CO_2 capture and separations. *Chemical Reviews*, 118(11):5488–5538, 2018. PMID: 29812911. arXiv:https://doi.org/10.1021/acs.chemrev.7b00691, doi:10.1021/acs.chemrev.7b00691.

[KM02] Martin Karplus and J Andrew McCammon. Molecular dynamics simulations of biomolecules. *Nat Struct Biol*, 9(9):646–52, Sep 2002. doi:10.1038/nsb0902-646.

[KPJB17] Mahzad Khoshlessan, Ioannis Paraskevakos, Shantenu Jha, and Oliver Beckstein. Parallel analysis in MDAnalysis using the Dask parallel computing library. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 64–72, Austin, TX, 2017. SciPy. doi:10.25080/shinma-7f4c6e7-00a.

[LJYH18] Denvid Lau, Wei Jian, Zechuan Yu, and David Hui. Nano-engineering of construction materials using molecular dynamics simulations: Prospects and challenges. *Composites Part B: Engineering*, 143:282 – 291, 2018. doi:10.1016/j.compositesb.2018.01.014.

[LS15] Chunyu Li and Alejandro Strachan. Molecular scale simulations on thermoset polymers: A review. *Journal of Polymer Science Part B: Polymer Physics*, 53(2):103–122, 2015. doi:10.1002/polb.23489.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth Jane Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comp Chem*, 32:2319–2327, 2011. doi:10.1002/jcc.21787.

[MM14] Cameron Mura and Charles E. McAnany. An introduction to biomolecular simulations and docking. *Molecular Simulation*, 40(10-11):732–764, 2014. arXiv:http://dx.doi.org/10.1080/08927022.2014.935372, doi:10.1080/08927022.2014.935372.

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

[Oro14] Modesto Orozco. A theoretical view of protein dynamics. *Chem. Soc. Rev.*, 43:5051–5066, 2014. doi:10.1039/C3CS60474H.

[PLK+18] Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, Goerge Chantzialexiou, Thomas E. Cheatham, Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Task-parallel analysis of molecular dynamics trajectories. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*, page Article No. 49, New York, NY, USA, August 13–16 2018. Association for Computing Machinery, ACM. doi:10.1145/3225058.3225128.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130–136, 2015. URL: https://github.com/dask/dask.

[Rot09] Jörg Rottler. Fracture in glassy polymers: a molecular modeling perspective. *Journal of Physics: Condensed Matter*, 21(46):463101, oct 2009. doi:10.1088/0953-8984/21/46/463101.

[SB14] Sean L Seyler and Oliver Beckstein. Sampling of large conformational transitions: Adenylate kinase as a testing ground. *Molec. Simul.*, 40(10–11):855–877, 2014. doi:10.1080/08927022.2014.919497.

[TCD+14] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, Sept.-Oct. 2014. doi:10.1109/MCSE.2014.80.

[The05] Douglas L Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallogr A*, 61(Pt 4):478–80, Jul 2005. doi:10.1107/S0108767305015266.

[Tuc10] Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford University Press, Oxford, UK, 2010.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

[VMMC+15] L. M. Varela, T. Méndez-Morales, J. Carrete, V. Gómez-González, B. Docampo-Álvarez, L. J. Gallego, O. Cabeza, and O. Russina. Solvation of molecular cosolvents and inorganic salts in ionic liquids: A review of molecular dynamics simulations. *Journal of Molecular Liquids*, 210:178–188, 2015. doi:10.1016/j.molliq.2015.06.036.

[Wic11] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 2011. doi:10.18637/jss.v040.i01.