Proceedings of the 19th Python in Science Conference

July 6 - July 12 • Austin, Texas

PROCEEDINGS OF THE 19TH PYTHON IN SCIENCE CONFERENCE

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

SciPy 2020
Austin, Texas
July 6 - July 12, 2020

ISSN:2575-9752
https://doi.org/10.25080/Majora-342d178e-02b

ORGANIZATION

Conference Chairs

JONATHAN GUYER, NIST
CORRAN WEBSTER, Enthought, Inc.

Program Chairs

GIL FORSYTH, CapitalOne
MATT HABERLAND, Cal Poly
NICOLAS HUG, Columbia University
PAUL IVANOV, Bloomberg
MADICKEN MUNK, University of Illinois

Communications

TANIA ALLARD, Microsoft
MATT DAVIS, Populus

Birds of a Feather

MATTHIAS BUSSONNIER, University of California, Merced
JULIE HOLLEK, Mozilla

Proceedings

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

Financial Aid

SCOTT COLLIS, Argonne National Laboratory
ERIC MA, Novartis Institutes for Biomedical Research
NADIA TAHIRI, Université de Montréal

Tutorials

ALEXANDRE CHABOT-LECLERC, Enthought, Inc.
MIKE HEARNE, USGS
SERAH RONO, The Carpentries

Sprints

RYAN MAY, University Corporation for Atmospheric Research
JUAN NUNEZ-IGLESIAS, Monash University
DHARHAS POTHINA, US Army Engineer Research and Development Center

Diversity

CELIA CINTAS, IBM Research Africa
MELISSA WEBER MENDONCA, Federal University of Santa Catarina

Activities

PAUL ANZEL, HEB
KYLE NEIMEYER, Oregon State University
INESSA PAWSON, Albus Code

Sponsors

JILL COWAN, Enthought, Inc.
KRISTEN LEISER, Enthought, Inc.

Financial

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

Logistics

JILL COWAN, Enthought, Inc.
KRISTEN LEISER, Enthought, Inc.

Proceedings Reviewers

ADRIAN HEILBUT
ALBERTO ANTONIETTI
ALEJANDRO WEINSTEIN
AMIR KHALIGHI
ANDREW REID
ANGELOS KRYPOTOS
ANIRUDH ACHARYA
BARGAVA SUBRAMANIAN
CALVIN MCCARTER
CHRIS CALLOWAY
CYRUS HARRISON
DANIEL CHEN
DAVID NICHOLSON
DAVID SHUPE
DHAVIDE ARULIAH
DILLON NIEDERHUT
FILIPE FERNANDES
GAJENDRA DESHPANDE
HIMEL MALLICK
HOMIN LEE
HONGSUP SHIN
HORACIO VARGAS GUZMAN
ISHA CHATURVEDI
IVAN MARROQUIN
JAIME ARIAS
KAY SUN
KELVIN LEE
KIRTAN DAVE
MARKUS ERWEE
MARTIN DURANT
MATT CRAIG
MATTHEW BENEDICT
MEGHANN AGARWAL
MICHAEL JOSEPH
MICHAEL SARAHAN
MIKE MCCARTY
NADIA TAHIRI
OLAV VAHTRAS
SAKET CHOUDHARY
SANDHYA PRABHAKARAN
SARVESH NIKUMBH
SCOTT SIEVERT
SERGE GUELTON
TZU-CHI YEN
YINGWEI YU
YU FENG

Accepted Talk Slides

Treating gridded geospatial data as point data to simplify analytics, Christine Smit, and Hailiang Zhang, and Mahabaleshwara Hegde, and Faith Giguere, and Long Pham
doi.org/10.25080/Majora-342d178e-019
Arkouda: Terascale Data Science at Interactive Rates, Benjamin Albrecht, and Michael Merrill, and William Reus, and Brad Chamberlain
doi.org/10.25080/Majora-342d178e-01a
Boost-histogram: High-Performance Histograms as Objects, Henry Schreiner, and Hans Dembinski, and Jim Pivarski, and Shuo Liu
doi.org/10.25080/Majora-342d178e-01b
Open-source bioimage analysis software to accelerate drug discovery, Anne Carpenter
doi.org/10.25080/Majora-342d178e-01c
cuSignal - GPU Accelerating SciPy Signal with Numba and CuPy, Adam Thompson, and Matt Nicely, and Graham Markall, and Brad Rees
doi.org/10.25080/Majora-342d178e-01d
Frictionless Data for Reproducible Biology, Lilly Winfree
doi.org/10.25080/Majora-342d178e-01e
Interactive Supercomputing with Jupyter at the National Energy Research Scientific Computing Center, Rollin Thomas, and Shane Canon, and Shreyas Cholia, and Matt Henderson, and Kelly Rowland, and Jon Hays, and William Krinsman, and Justin Ley, and Labanya Mukhopadhyay, and Trevor Slaton
doi.org/10.25080/Majora-342d178e-01f
Project Mjolnir: A Modular, Open-source Platform for Developing Scientific IoT Sensor Networks, C.A.M. Gerlach
doi.org/10.25080/Majora-342d178e-020
Pandera: Statistical Data Validation of Pandas Dataframes, Niels Bantilan
doi.org/10.25080/Majora-342d178e-021
Molecular infrastructure for modeling viruses with pythonic-mediated packages: pyF4all, Horacio V. Guzman
doi.org/10.25080/Majora-342d178e-022
pyhf: a pure Python statistical fitting library with tensors and autograd, Matthew Feickert
doi.org/10.25080/Majora-342d178e-023
Bringing GPU Support to Datashader: A RAPIDS Case Study, Jon Mease
doi.org/10.25080/Majora-342d178e-024
Learning from evolving data streams, Jacob Montiel
doi.org/10.25080/Majora-342d178e-025
Spatial Algorithms at Scale with spatialpandas, Dharhas Pothina, and Kim Pevey, and Adam Lewis
doi.org/10.25080/Majora-342d178e-026

Accepted Posters

Decentralized, Deterministic Robot Swarm Control using Blob Methods for PDEs, Matt Haberland, and Katy Craig, and Karthik Elamvazhuthi, and Olga Turanova
doi.org/10.25080/Majora-342d178e-018

SciPy Tools Plenaries

HoloViz: What's new and what's next, James A. Bednar
doi.org/10.25080/Majora-342d178e-028
SciPy Tools Plenary on Matplotlib, Elliott Sales de Andrade
doi.org/10.25080/Majora-342d178e-029
SciPy Tools Plenary on Numba, Siu Kwan Lam
doi.org/10.25080/Majora-342d178e-02a

Lightning Talks

Building an AutoML System for Fun and Non-profit, Niels Bantilan
doi.org/10.25080/Majora-342d178e-027

SCHOLARSHIP RECIPIENTS

OPETUNDE ADEPOJU, Venture Garden Group
DANIEL ALTHVIZ MORÉ, Spyder
ROHIT ARORA, University of Texas at Austin
SHREYAS BAPAT, Indian Institute of Technology Mandi
WAMBA TCHINDA CLAUDIN, UNIVERSITY OF YAOUNDE 1
CHRISTOPHER CURRIN, IBRO-Simons Computational Neuroscience Imbizo
SAYANTAN DAS, None
GAJENDRA DESHPANDE, KLS Gogte Institute of Technology, Belagavi
KADAMBARI DEVARAJAN, University of Massachusetts at Amherst
DIANA DIAZ, Wayne State University
ATANAS DOMMO, University of Yaounde 1
GEZEHAGN GUTEMA EGGI, Arsi University
SAM FRIEDMAN, Texas A&M University
JAMIL GAFUR, Los Alamos National Lab
BYRON GEOFFREY, Nova Southeastern University
JUANITA GOMEZ, Spyder IDE
MAXWELL GROVER, University of Illinois at Urbana-Champaign
STEPHANNIE JIMENEZ GACHA, Spyder IDE
ESHIN JOLLY, Dartmouth College
SALOMON KABONGO KABENAMUALU, African Master in Machine Intelligence, University of Ghana
JOSHUA KALOGNIA, Council for scientific and Industrial Research-Institute for Scientific and Technological Information
TETSUO KOYAMA, GetFEM++
GUILHERME LEOBAS, Quansight - Numba
EDGAR ANDRÉS MARGFFOY TUAY, Spyder IDE
KRYSTAL MAUGHAN, University of Vermont
ABIGAIL MCGOVERN, Monash University
JESSICA MEJIA, University of South Florida
AMBER NADEEM, Zilltech.net
EMILIAN NGATUNGA, University of Dodoma
SOLOMON NSUMBA, Makerere University AI and Data Science lab, Uganda
ESTHER ODUNTAN, African Institute of Mathematical Sciences
OLAIDE OJOMO, University of Texas at Austin
AKSHAY PAROPKARI, University of California, Merced
WAISWA PHILIP, Uganda Technology and Management University
MRIDUL SETH, econ-ark
SHUBHAM SHARMA, Geospoc
SCOTT SIEVERT, University of Wisconsin–Madison
HORACIO VARGAS GUZMAN, Institute Josef Stefan, Slovenian Research Council
YUE WU, University of Georgia
SUBHADITYA MUKHERJEE, None
ERMIYAS BIRHANU BELACHEW, None
TCHAMBOU TCHOUONGSI LANDRY, None
ZAC HATFIELD DODDS, Australian National University
LAUREN BIERMANN, Plymouth Marine Laboratory

CONTENTS

Preface

Meghann Agarwal[¶], Julie Hollek[§], Dillon Niederhut[‡*]

SciPy 2020, the 19th annual Python in Science Conference, was held July 6-12, virtually via the conference platform Crowdcast. Due to the COVID-19 pandemic, the SciPy conference was held online. The SciPy Conference brings together a community of researchers, engineers, and programmers dedicated to the advancement of scientific computing through open source Python software.

The two main conference themes for 2020 were high performance computing; and, machine learning and data science. Discipline-specific symposia included astronomy and astrophysics; biology and bioinformatics; materials science; earth, ocean, geology, and atmospheric science; and a new symposium dedicated to fostering conversations among maintainers of the open source infrastructure that help power the worlds of scientific discovery and engineering. As was the case in 2019, there were plenary sessions for updates about key scientific software libraries, and three sessions of the ever-popular lightning talks, which this year included SciPy's youngest speaker, Artash Nath, discussing machine learning approaches in exoplanet research.

The first conference keynote lecture was delivered by Anne Carpenter, who discussed the history of CellProfiler in the context of developing academic software, current application of the scientific software stack to problems in biology, and future directions for tasks like drug discovery, powered by machine learning. Andrew Chael delivered the second keynote, about the large, inter-organizational effort to take the first photograph of black hole M87, and the role of scientific software in that project. This year's diversity plenary was given by Bonny McClain, who delivered an interactive lecture about bias in data, and how to think about measuring what people haven't thought about measuring before.

The online format permitted a larger-than-usual number of participants, ultimately attracting 1412 participants from a record-breaking 57 countries, making this the largest SciPy Conference yet. Participants reported that they enjoyed the broader access to beginner tutorials for popular libraries like PyTorch and xarray -- something that would not be possible without having the conference at least partially online. Birds of a Feather (BoF) sessions were organized around the topics of packaging, diversity, Python in education, hardware, and SciPy 2021 with great attendance due to the online format. Sprints that usually gather around tables in conference rooms took the conversation to virtual tables using a variety of technologies for text and voice chat. The online format did come with its own set of challenges, in particular, promoting serendipitous conversations that are typical in the "hallway track" at the event along with the typical audience interaction seen in BoF sessions and groups of participants trekking in the Austin heat to enjoy tacos or other fine Austin fare and each others' company.

Of this year's conference attendees, 22% identify as women, continuing SciPy's trend in increasing participation of people from minoritized communities. The organizers identified LGBTQ, African American, Native American, Middle Eastern, and Hispanic/Latinx scientists as still underrepresented at the conference, and targets for future equity and inclusion efforts. With this in mind, SciPy announced a new initiative to provide additional scholarship funding for Black, Indigenous, and People of Color (BIPOC) to attend the conference, starting in 2021.[1]

While the global circumstances have been disruptive to all facets of life, their effect on the conference was greatly mitigated, largely due to the superhuman efforts of Jill Cowan and Kristen Leiser. In particular, Jill started organizing for SciPy in 2014, and over time has become the heart of the conference. Attendees regularly remark that SciPy is the most open and friendly conference that they attend, and typically add that they recall that the first moment they felt this way was upon meeting Jill at the registration desk. To add an editorial note, the SciPy Conference would not be where it is today without Jill's leadership over the last six years; and, our own efforts as committee chairs have been made significantly lighter due to her hard work. Jill is leaving the conference this year for a well-deserved retirement, but she will always be remembered in the community that she helped build.

We dedicate this work, the 19th Python in Science Conference Proceedings, to Jill Cowan.

On behalf of the SciPy 2020 organizers,

Meghann Agarwal
Julie Hollek
Dillon Niederhut

¶ *Oracle*
§ *Mozilla*
* *Corresponding author: dillon.niederhut@gmail.com*
‡ *Novi Labs*

1. The full statement is available at https://www.scipy2020.scipy.org/support-of-black-communities

Securing Your Collaborative Jupyter Notebooks in the Cloud using Container and Load Balancing Services

Haw-minn Lu‡*, Adrian Kwong‡, José Unpingco‡

Abstract—Jupyter has become the go-to platform for developing data applications but data and security concerns, especially when dealing with healthcare, have become paramount for many institutions and applications dealing with sensitive information. How then can we continue to enjoy the data analysis and machine learning opportunities provided by Jupyter and the Python ecosystem while guaranteeing auditable compliance with security and privacy concerns? We will describe the architecture and implementation of a cloud based platform based on Jupyter that integrates with Amazon Web Services (AWS) and uses containerized services without exposing the platform to the vulnerabilities present in Kubernetes and JupyterHub. This architecture addresses the HIPAA requirements to ensure both security and privacy of data. The architecture uses an AWS service to provide JSON Web Tokens (JWT) for authentication as well as network control. Furthermore, our architecture enables secure collaboration and sharing of Jupyter notebooks. Even though our platform is focused on Jupyter notebooks and JupyterLab, it also supports R-Studio and bespoke applications that share the same authentication mechanisms. Further, the platform can be extended to other cloud services other than AWS.

Index Terms—data science, infrastructure, jupyter, rstudio

Introduction

This paper focuses on secure implementation of Jupyter Notebooks and Jupyter Labs in a cloud based platform and more specifically on Amazon Web Services (AWS) though many architectures and methods described here are applicable to other cloud platforms. As Jupyter is the mainstay of scientific programming in python, the ability to analyze data in a secure environment enables the researcher to access data that is either sensitive or encumbered by compliance to regulations such as Health Insurance Portability and Accountability Act (HIPAA) which might otherwise not be available.

Security is paramount for applications that process sensitive data in areas such as defense, finance, and healthcare. Broadly speaking, security regulations can be characterized in terms of authentication (verifying the credentials of users and their access to resources), encryption (data is encrypted at rest and in transit), auditing (providing surveillance of key resources) and vulnerability mitigation (antivirus and security updates).

In the architecture section, we describe how our architecture using AWS Elastic Container Service (ECS) facilitates encryption at-rest and in-transit and integrates an application load balancer (ALB) for authentication.

In the Applications and Authentication section, we dive into the details of the ALB and how JSON Web Tokens (JWT) facilitate integration with Jupyter and RStudio.

In the Security and Compliance section, we address the encryption of the underlying cloud architecture, auditing capabilities, and mitigation of vulnerabilities.

Our specific implementation satisfies privacy and security concerns and can serve as a starting point to develop customized solutions for related use cases.

Background

To implement a cloud based Jupyter compute platform is not difficult. Project Jupyter includes Jupyter Hub, which provides a proxy and container management. In particular the Zero to Jupyter Hub with Kubernetes project, [Pro20] provides a framework to implement Jupyter Hub on a Kubernetes platform. It is intended as a quick method to deploy a cluster of Jupyter notebooks or Jupyter labs easily. However, it has many significant drawbacks.

First, Kubernetes is notoriously difficult to secure and has many vulnerabilities that are not addressed by default as evidenced by these recommendations for securing a Kubernetes cluster [Mar18]. One of the primary reasons is Kubernetes is immensely complex.

Second, Zero to Jupyter Hub with Kubernetes to date does not have a simple solution to the problem of encryption in transit (encryption of all data over a network). All proposed solutions (e.g., `istio` or `weave`) rely on yet another overlay in Kubernetes making the solution even more complicated.

Literature on securing Jupyter in the cloud is scant but solutions to individual issues can sometimes be found searching through blogs, `github` issues and help sites. In this paper, the reader is given a solution that can meet most security concerns in one place, while not placing an undue burden on the end user or the system administrator. The mantra of *security through simplicity* is adopted.

Architecture

There are two distinct levels of architecture described. The cloud architecture comprises the various cloud services which is the lower layer of virtualization. The container architecture is the top layer virtualization built on top of the cloud architecture.

* *Corresponding author: hlu@westhealth.org*
‡ *Gary and Mary West Health Institute*

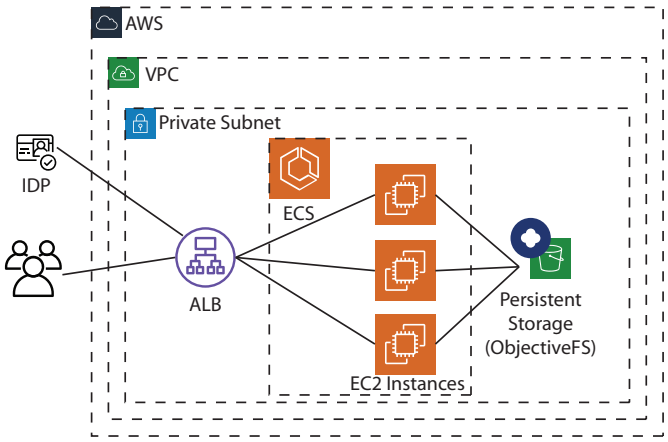


Fig. 1: *Cloud Architecture.*

Cloud Architecture

The basic cloud architecture is shown in Fig. 2. It consists of an identity provider (IDP) used to authenticate a user, an application load balancer (ALB) to regulate user access through authentication, and a fleet of elastic cloud computer compute (EC2) instances to instantiate the containers. Finally, ECS manages the containers deployed on the EC2 cluster.

Elastic Container Service: ECS is a container orchestration service. A container is instantiated as an ECS *task*. ECS provides a resource called a *task definition* that allow for the configuration of the container image, the environment variables, command override and container port.

Taking the most naive approach, ECS can be instructed to start a task based on a task definition. After the task has fully started, the host among the EC2 instances and the mapped port (the port on the EC2 node which is mapped to the container port) is known. At this point, one could write a monitoring function to detect when a task has started, retrieve the specific host and mapped port and create a listener rule for the application load balancer.

Instead of this cumbersome procedure, ECS provides another resource called a *service*. A service can manage many aspects of tasks within ECS including the number of tasks and a *target group* associated with the service. For our purposes managing the number means selecting a desired count of 1 or 0 depending on whether the container is running or has been automatically culled due to inactivity. A target group is a collection of host ports or serverless AWS Lambda functions, to which a listener rule can direct network traffic. In short by specifying a target group to a service, the host and mapped port are automatically assigned to the target group when a task has fully started.

Application Load Balancer: AWS's ALB can comprise multiple listeners to support multiple protocols. To maintain security, enforcement of HTTPS should be maintained either by not including a listener for HTTP or providing an HTTP listener that redirects all requests to HTTPS.

AWS's ALB, through a listener, is able to direct external HTTPS requests to various components. Based on listener rules, a request can be directed on the basis of both the hostname and the path. As an example, we use a path to specify a user and a service such as Jupyter (for example, `domain.com/user_id/Jupyter` or `domain.com/user_id/rstudio`), this allows us to give each user their own container.

Each listener rule maps a path, hostname or both to a particular target group. Since we use an ECS service, we can assign a particular service to a target group. The service then manages which ports and EC2 instances are part of the target group.

While the ALB can enforce encryption from the end user to the ALB, the container application (e.g., Jupyter) should also be configured to listen only for HTTPS. In this manner, the communication from the end user to the ALB is encrypted as is the communication from the ALB to the container application, ensuring end to end encryption.

Furthermore the application load balancer is also configured to perform authentication from an OpenID Connect (OIDC) compliant IDP. This eliminates the need for multiple messages to be passed when using either SAML or OAuth. Upon authentication, the ALB attaches three fields to the header of the http request `x-amzn-oidc-accesstoken`, `x-amzn-oidc-identity` and `x-amzn-oidc-data` which can be used by the end application to confirm the user's identity and validate the authentication. An example of this process as implemented in a Jupyter notebook is described below.

For our IDP, we use Okta since it allows us to federate identity services to additional sign on services. This allows us to onboard collaborators and allow the collaborators to manage their users.

Shared Storage: In order to facilitate persistence across containers and also collaboration, ECS orchestrates containers on EC2 instances instead of AWS's Fargate product (Fargate facilitates containers in a serverless fashion but does not provide a host to mount an ObjectiveFS file system). Persistent storage can be mounted on the underlying EC2 instances. Individual containers can access the persistent storage by bind mounting the persistent storage. To meet security compliance of encryption at rest, the persistent storage should be encrypted. We elected to use the third party ObjectiveFS for cost reasons though native AWS resources such as elastic file system (EFS) can be used provided that both the file system and the network communications to the file system are encrypted. [Ser20c] ObjectiveFS is a secure file system backed by AWS simple storage service (S3). It should be noted to meet encryption in transit compliance requirements that any network attached storage must have network communications encrypted. For example, the base network file system (*nfs*) protocol is not.

As a specific example with Jupyter notebooks we mount persistent storage as `/media/home/`. For a given user say `user_a` we bind mount `/home/jovyan` to `/media/home/user_a` so that while in the container the user sees `/home/jovyan` the home directory the users files are actually stored in the persistent storage in a `user_a` subdirectory. This configuration has two advantages. Only one persistent volume is needed to support all users' home directories minimizing costs and within the container all users see /home/jovyan thus eliminating the need to build a separate Jupyter container image for each user.

With this configuration, multiple services can use the same home directory. For example, in our R Studio deployment `/home/rstudio` is also mapped to `/media/home/user_a`. Furthermore, we also can provide a persistent volume for shared directories. For example, for all users on `project_a` we bind mount `/home/jovyan/projects/project_a` to `/media/projects/project_a` where the persistent volume is mounted to `/media/projects`.

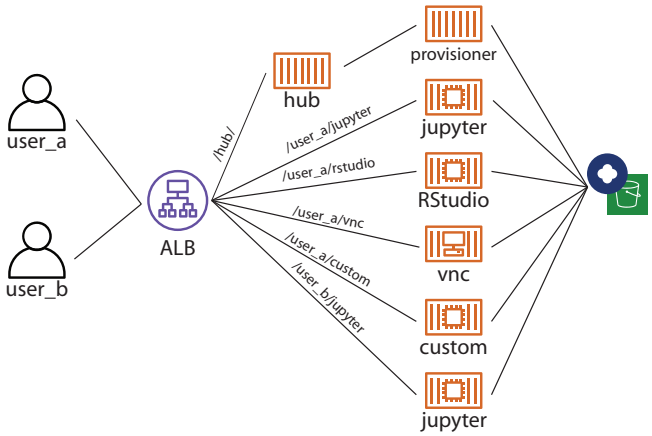


Fig. 2: *Cloud Architecture.*

Resource Summary: To securely implement the above cloud architecture, each container instance for each user has a set of resources associated with it. First, a task definition is created for each user, this enables customized bind mounts as described above. Additionally, custom environment variables or task commands can also be supplied through the task definition. The task definition can also direct logging the the appropriate AWS CloudWatch stream.

Each user also has a ECS service, ALB listener rule and target group associated with it. This allows the seamless management of connecting a user to the desired container instance.

Finally each service has an AWS IAM role associated with it, this ensures the user has only the access rights to our AWS cloud that are need by the user. Beyond the rights to operate the container task, additional rights might include access to certain S3 storage or certain AWS Secrets Manager. As an example, we use the AWS Secrets Manager to manage user's credentials to various databases and public/private keys.

To simplify management of the per user resources, an AWS CloudFormation template is used to ensure consistency and uniformity among cloud resources whenever a new container instance/user combination is spun up. As an example, our CloudFormation template contains an IAM role, listener rule, target group, task definition, and an ECS service. Each template is then customized to spin up a CloudFormation stack for each user and application combination.

Container Architecture

The architecture in terms of container comprises a persistent hub container, an optional ephemeral provisioner container, and an assortment of semi-persistent application containers such as Jupyter notebook. In an alternative deployment, AWS Lambda functions can be functionally substituted for the hub container, but for the sake of simplicity only the container version of the hub is described.

The application containers are described as semi-persistent as they can be started on demand and culled when one or more inactivity criteria has been reached. This can be achieved by updating the associated service to have a desired count of `1` to start or a desired count of `0` to cull.

We adopted a url path routing convention to access each application such as `domain.com/user_id/application`

Container Management: The heart of the system is the hub container. To facilitate ALB authentication, two listener rules are provided. One rule allows anyone to connect, so that the hub can present a login page (with single sign on and IDP this looks like a single login button). The login action redirects the browser to a url which forces authentication via the ALB. Though this step is not necessary, it provides a cue that makes for a smoother user experience.

Since the hub container may be given privileges to set IAM roles for the application services, the role under which the hub service runs can have a boundary policy attached to it [Ser20d]. This ensures that any role created by the hub service is constrained to include the boundary policy. This prevents the hub from being able to create an arbitrary role should the container become compromised.

The provisioner container is an ephemeral task which is run with the persistent storage mounted. The provisioner can create a home directory for a user the first time the user logs in and provision the directory with any necessary files. While the functionality of the provisioner container could be incorporated in the hub container. Separation allows the provisioner to run with minimal cloud privileges (IAM role) and allows the hub to have no access to the shared home directory, so in the event the hub container is compromised the user's file system is not exposed. Also, with separation the hub does not have to have access to the file system so it can be refactored and deployed as a Lambda function. Furthermore the provisioner container runs very briefly further limiting the vulnerability window.

Once authenticated, the user can elect to connect to an application container. This can occur under three circumstances: the user's application container is still running, the user's application container has been culled, or the user has never started the application before. If the container is still running, the user is immediately redirected to the container. If the container has been culled, the service is updated to a desired count of `1`. If the application has never been started by the user, resources to spin up the service are created such as by creating a CloudFormation stack.

Additionally, an option to "decommission" an application can be presented where the CloudFormation stack can be deleted.

Culling: The best practice for culling an application is to have the application upon exiting, set the desired count to `0` of its corresponding service.

For the example of Jupyter, the start up scripts for both Jupyter notebook and Jupyter lab contains the following snippet with `main` imported from different places:

```
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$',
                         '', sys.argv[0])
    sys.exit(main())
```

Rather than just exiting after `main` completes, a modified start up script updates the desired count of the corresponding service to `0`. Since `boto3` essentially wraps API calls to AWS, a delay before termination is needed to ensure the update API call is received before terminating the task. Failure to change the desired count will only result in the service restarting the container upon termination.

```
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$',
                         '', sys.argv[0])
    main()
    session = boto3.Session()
```

```
    ecs = session.client("ecs", region_name)
    ecs.update_service(cluster=cluster_name,
                       service=service_name,
                       desiredCount=0)
    # Sleep for 2 minutes give service time to update
    time.sleep(120)
```

Code to retrieve the `region_name`, `cluster_name`, and `service_name`, are omitted for clarity, but they can be retrieved from environment variables (set in task definition), passed via `sys.argv` or even by calls to `boto3`. Though the first two options are simpler.

The above modification to the start up scripts ensures that when Jupyter exits the task count is zero. However, in order for this to be meaningful culling parameters in the Jupyter configuration such as `c.NotebookApp.shutdown_no_activity_timeout` `c.MappingKernelManager.cull_connected`, `c.MappingKernelManager.cull_idle_timeout` and `c.MappingKernelManager.cull_interval`, as well as setting a shell timeout (e.g., `TMOUT` environment variable are set) in the event a terminal is open.

Authentication and Applications

As mentioned above, the bulk of the authentication is performed by the ALB. However, it is important for the individual application to validate a request forwarded by the ALB, for two reasons. Validation prevents potential security vulnerabilities due to a misconfiguration in the system or exposes security vulnerabilities during the initial system debugging. Additionally, validation ensures that the identity of the user is what is expected. The ALB ensures that the user has validly authenticated, but it is up to the application to ensure that the correct user has connected.

Validation is achieved through the JWT token presented in the `x-amzn-oidc-data` header by the ALB. These JWT tokens are signed by a public key retrievable from AWS ensuring that only the ALB could have signed them. Within the JWT token, the `kid` field represents the *key ID* for the public key. To validate, the key ID should be extracted and corresponding public key should be retrieved from AWS. With the public key, the JWT token can then be validated. We use the `python-jose` module available on PyPi. The `sub` field in the JWT token is the same as the OIDC ID which is also presented in the `x-amzn-oidc-identity` field. The application should then verify this is OIDC ID associated with the expected user.

To deploy an application securely in our infrastructure, in addition to validating the authentication, the application container should meet four more requirements. It should have a configurable base url as the ALB will forward requests to the application with the base url prefix. It should communicate to the ALB over HTTPS to ensure end to end encryption. It should provide a url to respond to pings sent by the ALB for health checks. It should validate that the mounted home container belongs to the user.

The solution to the last requirement is for our provisioner to write an `.id` file in the user's home directory containing the user's ID. This file is written by `root` and is only readable. The application upon startup or authenticaation can verify that the user has the correct home directory mounted. This requirement is a safeguard against misconfiguration and can be omitted if one is confident that the system is not misconfigured.

Jupyter

Implementing authentication for Jupyter notebook/lab is particularly challenging as they do not combe with a pluggable authentication module, unlike JupyterHub. In order to implement validation, the source file `login.py` must be modified. This file is usually located in the `notebook/auth/` directory in your `site-packages` or `dist-packages` directory. Since Jupyter notebook and JupyterLab are not truly separate applications (in fact they are interchangeable using the path `/tree` or `/lab`), the same `login.py` file facilitates authentication for both. If you build using a standard docker image such as `jupyter/base-notebook` or any of its derivative notebooks, this directory would be `/opt/conda/lib/python3.x/site-packages` directory. Please note that the specific python version may vary dependent on which version of the docker container is used and whether subsequent additional install modules might force a rollback of python versions.

The specific modification to the `login.py` file involves replacing two methods, the `get` method and the `get_user_token` class method of the `LoginHandler` class.

Unaltered, the method `get` determines whether the `current_user` is set indicating the user has been logged in. If not authenticated, the function presents a login page. Our modification simply adds an additional check that if `current_user` is not set, we validate the JWT token in header to determine additionally whether the user is authenticated. It should also be noted that the function is also decorated as a coroutine to make the function asynchronous as the verification may require network access to retrieve a public key.

```
@tornado.gen.coroutine
def get(self):
    authenticated = False
    if self.current_user:
        authenticated = True
    else:
        if self.verify_jwt():
            authenticated=True
    if authenticated:
        next_url = self.get_argument('next',
            default=self.base_url)
        self._redirect_safe(next_url)
    else:
        self._render()
```

The other method to be replaced is the `get_user_token`. Unaltered, the method returns the authorization token used as part of a notebook/lab minimal authentication scheme. This token is normally supplied as a query string in the URL or through the login page. We bypass this mechanism altogether. Instead, we examine the request header for a JWT token supplied by AWS and validate it. If it is successful we provide a token. As far as the rest of the notebook code the value of the token is not used so we supply a random string. Our version of `get_user_token` uses a local cache to store retrieved public keys and previously the previously decoded user ID.

```
@classmethod
def get_user_token(cls, handler):
    """Identify the user based on
       Authorization header

    Returns:
    - uuid if authenticated
    - None if not
    """
```

```
    authenticated = False
    if cls.verify_oidc(handler):
        authenticated = True
    else:
        oidc_jwt = handler.request.headers\
            .get('x-amzn-oidc-data')
        if oidc_jwt:
            try:
                header = jwt.get_unverified_headers( \
                    oidc_jwt)
            except JOSEError:
                return None
            kid = header.get('kid')
            if kid and kid == user_cache.get('kid') \
                and user_cache.get('pk'):
                try:
                    token = jwt.decode(oidc_jwt,
                                       user_cache['pk'])
                except JOSEError:
                    return None
                oidc_id = handler.request.headers\
                   .get('x-amzn-oidc-identity')
                if token['sub'] == oidc_id:
                    authenticated = True
                    user_cache['jwt'] = oidc_jwt
                    user_cache['user_id'] = oidc_id
    if authenticated:
        return uuid.uuid4().hex
    else:
        return None
```

In addition to the two modified methods, we supply two helper methods `verify_jwt` for `get` and `verify_oidc` for `get_user_token`. They perform the token validation and cache management. Additional code which can read identifiers in persistent volumes and verify they match the user who is authenticated can also be added to ensure two authenticated users don't have access to the other's containers.

```
def verify_jwt(self):
    global user_cache
    oidc_id = self.request.headers\
                .get('x-amzn-oidc-identity')
    oidc_jwt = self.request.headers\
                .get('x-amzn-oidc-data')

    if not oidc_jwt:
        self.log.warning("No JWT Token in Header")
        return False

    if (user_cache.get('user_id') == oidc_id and \
        user_cache.get('jwt') == oidc_jwt):
        return True

    try:
        header = jwt.get_unverified_headers(oidc_jwt)
    except JOSEError as e:
        self.log.error("JWT failed to decode: {}"\
                    .format(e))
        return False

    kid = header.get('kid')
    if not kid:
        self.log.error("No Key ID in JWT token")
        return False

    if kid != user_cache.get('kid'):
        if 'pk' in user_cache:
            del user_cache['pk']

    if not 'pk' in user_cache:
        try:
            r = requests.get(PK_SERVER + kid)
            # TODO treat return code
            user_cache['pk'] = r.text
            user_cache['kid'] = kid
        except requests.RequestException as e:
            self.log.error("Requests Error: {}"\
                        .format(e))
            return False

    try:
        token = jwt.decode(oidc_jwt,
                           user_cache['pk'])
    except JOSEError as e:
        self.log.info("JWT failed to validate: {}"\
                    .format(e))
        return False

    if token['sub'] != oidc_id:
        self.log.error("User ID in token doesn't "
                       "match user ID in header")
        return False

    user_cache['user_id'] = oidc_id
    user_cache['jwt'] = oidc_jwt

@classmethod
def verify_oidc(cls, handler):
    global user_cache
    oidc_id = handler.request.headers\
                .get('x-amzn-oidc-identity')
    oidc_jwt = handler.request.headers\
                .get('x-amzn-oidc-data')

    if not oidc_id or not oidc_jwt:
        return False
    if oidc_id != user_cache.get('user_id'):
        return False
    if oidc_jwt != user_cache.get('jwt'):
        return False
    try:
        header = jwt.get_unverified_headers(oidc_jwt)
    except JOSEError:
        return False
    kid = header.get('kid')
    if kid != user_cache.get('kid'):
        return False

    return True
```

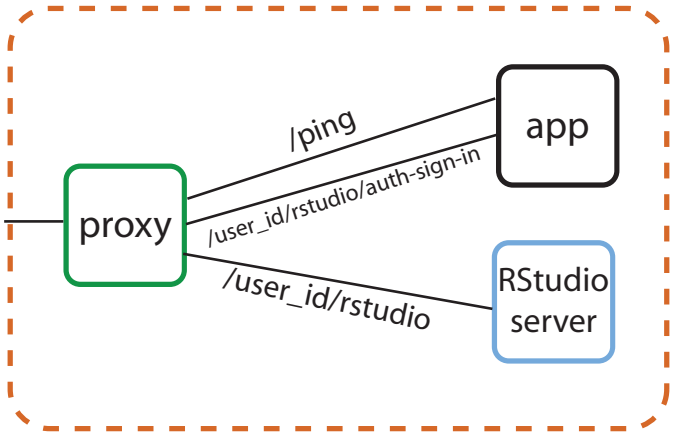


Fig. 3: *Inside the RStudio Container*

To meet the other requirements for Jupyter, the `base_url` configuration needs to be set to ensure that the route is properly interpreted. Furthermore, we use this `base_url` as the health check url which responds with a `302` code. A self-signed certificate is automatically generated when the container starts and that certificate is then used to configure Jupyter to run over HTTPS.

RStudio

Our implementation of RStudio Server on the same cloud platform is non-invasive to the code base, but more complicated architecturally. Since RStudio does not have a way to set the base URL of the application, a proxy is required to rewrite the HTTPS request paths. We use an `nginx` proxy to rewrite requests to RStudio Server using the `proxy_redirect` directive.

Figure shows the application structure within the RStudio container. A proxy communicates with the ALB and routes some requests to a custom app used for authentication and handling the health checks and others to the RStudio server. Since communications between the proxy, app and RStudio server are all within the container and not exposed, they do not require encryption to satisfy compliance. A self-signed certificate is created upon container startup that enables `nginx` to communicate over HTTPS to the ALB.

For authentication, RStudio Server maintains authentication session information in a cookie. So with `nginx` we capture, the `auth-sign-in` URL and redirect it to an lightweight webapp whose sole function is to authenticate the user, set the cookie and redirect the browser to RStudio Server. Since the app is necessary in this configuration, we also configure the app to respond to a `/ping` request issued by the ALB target group's health check.

The authentication code is nearly identical to the `verify_jwt` function written above for Jupyter. The cookie consists of three pieces, a user ID (which we retain as the default `rstudio` as we retained `jovyan` for the Jupyter notebook, to prevent the need to build a separate docker image for each user), the expiry and an HMAC 256 signature, signed with a secret typically stored at `/var/lib/rstudio-server/secure-cookie-key` inside the container. The following snippet of code implements this.

```
from urllib.parse import quote
from Crypto.Hash import HMAC
from Crypto.Hash import SHA256
import base64
import datetime

utc = datetime.datetime.utcnow()
expiry = utc + datetime.timedelta(days)
now = expiry.strftime('%a, %d %b %Y %H:%M:%S GMT')
dig = base64.b64encode( \
        HMAC.new(secret,
                 "{0}{1}".format(username, now),
                 digestmod=SHA256).digest())

cookie = quote("{0}|{1}|{2}".format(username,
                                    now,
                                    dig.decode()),
               '|')
response.set_cookie('user-id', cookie)
```

The `days` is the number of days til the cookie expires, and `username` is the user name (i.e. `rstudio`). In the above snippet, the cookie is attacked to a Flask response.

Virtual Network Computing (VNC) Containers

There are many desktop apps for Linux which may also be useful to deploy via a web application on a cloud cluster such as presented here. The following implementation allows the deployment of such applications such as Orange and Falcon through the use of a web VNC client to a VNC server running in a container.

This is based on the Docker Headless VNC Container project [Con19] as a blueprint using the `xfce4` window manager. Since it appears that the project has been inactive for over a year we adopt its `Dockerfile` as a starting point but do not use the docker images as a building block.

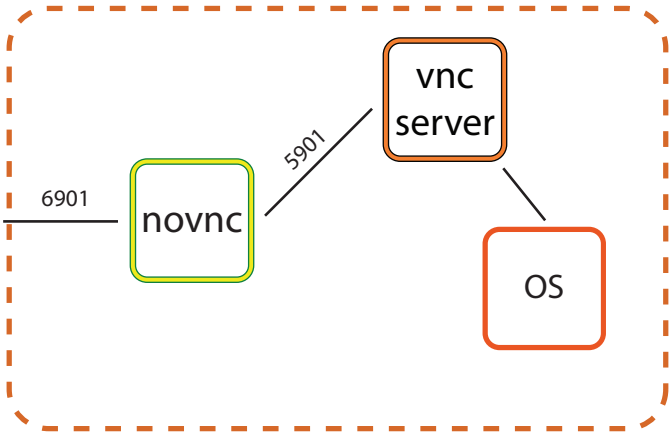


Fig. 4: *Inside a VNC Container*

Figure 4 shows the application structure within a headless VNC container. NoVNC [noV20] is used as a web to vnc proxy which connects via VNC to a local vnc server which in accordance to the Docker Headless VNC Container project is tigerVNC [Tig20]. Through the VNC server graphically oriented operating system commands and applications can be executed. In our container tigerVNC is unchanged and is installed just as it is in the headerless project's `Dockerfile`. The noVNC project comprises a `novnc` and `websockify` component. No changes were made to the `novnc` component except to alter the parameters use to start `websockify`. Therefore the focus of the customization is on the `websockify` component.

Fortunately, `websockify` permits authentication plugins. The plugin is a simple class with an `authenticate` method which accepts the `headers`, `target_host` and `target_port` as parameters. Upon success it returns and on failure it raises an `AuthenticationError` exception. Since the body of the code is essentially the same as the *verify_jwt* method descirbed for Jupyter, the code is not repeated here.

It should be noted that in the container by default the VNC server listens on port 5901 and the novnc client listens on port 6901. It is recommended that only port 6901 be exposed so that only the novnc client can directly communicate with the VNC server as the VNC password in this environment is not well protected. By only exposing port 6901, knowledge of the VNC password can not be exploited to bypass the authentication.

Furthermore, the web server within the `websockify` project is located in `websockifyserver.py` and is based on `SimpleHTTPServer`. It may be desirable to create a custom handler or custom `do_GET` method to handle issues such as providing a base URL, health check URL for the ALB's target group, or to implement templating if desired. A self-signed certificate is generated in a *launch.sh* as *self.pem* which the webserver will automatically detect and run using HTTPS.

Once this base container image is built with those customizations. Applications such as Orange or Falcon can be added, thus not limiting the cloud system to web applications.

Custom Applications

In developing your own bespoke applications, a layer of authentication can be employed. In consideration of developing or adapt-

ing your own application, you should provide an unauthenticated URL for the ALB's health check and be equipped to configure the base URL. Authentication can be easily plugged into most web server frameworks.

As a simple example, using flask authentication can be incorporated into a custom `login_required` decorator, so that for any protected URL the request is authenticated before being processed. Once again the decorator could be implemented with code similar to that of *jwt_verify* described above.

Security and Compliance

In our cloud architecture, the bulk of the security and compliance is built into the EC2 instances serving as nodes behind the ALB and built into features of the ALB. By keeping most of the security external to the containers, container images need less customization for security purposes making it easier to support a wide variety of container images and container apps.

The preferred method to implement security, compliance, and even maintenance services on an EC2 instance is to install the appropriate software in an Amazon machine image (AMI). By building a customized AMI based off an optimized Amazon ECS reference AMI [Ser20b] but including the desired additional services installed, an fully equipped EC2 instance can be spun up quickly and features such as autoscaling can easily be applied.

Specifics to security and compliance implementations are described in the following subsections including encryption at rest, access controls, auditing and other agents.

Encryption at Rest

As previoiusly mentioned, persistent storage and associated file system protocol are encrypted give both encryption in transit and encryption at rest for the persistent storage. However, it is also important that the base file system of the EC2 instances are also encrypted to fully ensure encryption at rest. There are two important aspects of ensuring encryption at rest for the base file system. First the attached file system such as elastic block storage (EBS) must be encrypted. This is accomplished by selecting encryption when creating the EC2 instance or within a launch configuration. Fortunately, AWS now offers an account-level option where EBS volumes are encrypted by default for any EBS volumes created in that given account. We highly recommend this option as it will mitigate the chances of misconfiguration.

Furthermore, the AMI used to create EC2 instances must also be encrypted. A common technique for doing so is to build an machine snapshot will all the agents and services desired then encrypt the snapshot. Regardless for what techinque is used. the AMI's should be encrypted to satisfy any requirements for encryption at rest.

Access Control

Another security concern is controlling the internet access from the container. The reason is two fold. First, controlling access allows us to prevent users from within a container from accessing potentially malicious websites. Second, should a container become compromised we want to mitigate the compromised container's ability to escalate privileges or pivot to other services within the organization. While AWS through the use of security groups and access control lists provide a coarse ability to regulate what destinations are accessible, we favor more fine grain control.

There are two aspects of this finer grain control, first we use an on-host firewall to control outbound access from the hosted containers. Second we funnel all traffic from each container to a proxy.

For the firewall, we use `iptables` using the following commands:

```
iptables --insert DOCKER-USER --in-interface docker0 \
   -o eth0 -j DROP

iptables --insert DOCKER-USER \
   --destination 169.254.169.254 --jump REJECT \
   --reject-with icmp-port-unreachable

iptables -t nat -A PREROUTING -i docker0 \
   -d 172.17.0.1 -p tcp --dport 8888 -j RETURN
iptables -t nat -A PREROUTING -i docker0 \
   -d 172.17.0.1 -p tcp -j DNAT --to-destination :2
```

The first command blocks all internet traffic coming from the *docker0* interface (where the containers must route through) to the *eth0* interface which is the external interface. The second command (see [Cos18]) blocks access to the node specific metadata service, which typical contains information about the EC2 instance and credentials for that instance.d. Blocking this prevents a compromised container from accessing the metadata about the EC2 instances blocking a potential escalation in privileges to that of the EC2 node. The third and fourth commands allows the container access to the EC2 instance (which in the docker world is IP address `172.17.0.1`) only on port 8888, where the proxy is configured to listen. All other access is routed to port 2 which has no active listeners.

On the container side, the environment variables `http_proxy` and `https_proxy` must be set to forward all http and https request to the EC2 instance at port 8888. In addition the `no_proxy` environment variable should be set to allow some traffic not to be forced into the proxy. Of course, `localhost` (and corresponding IP address `127.0.0.1`) do not require proxy as the traffic doesn't leave the container. In addition, the metadata IP address `169.254.169.254` should be allowed out so that the `iptables` rule regarding the metadata traffic can be enforced. Finally, the IP address `169.254.169.2` is used by the ECS agent.

Two methods can be used to address the environment variables. Either we can add the environment variables to the task definition when an application service created or it can defined in the container's *Dockerfile* with the following lines:

```
ENV http_proxy=http://172.17.0.1:8888/
ENV https_proxy=http://172.17.0.1:8888/
ENV no_proxy=localhost,127.0.0.1,\
169.254.169.254,169.254.170.2
```

Because of the `iptables` rules a misconfiguration that fails to set the proper environment variables results in loss of access and not a vulnerability.

The proxy can then determine whether to route the connection request directly externally or through an external outbound gateway which could include a company firewall so that broad based policies could be applied. For the proxy we selected `tinyproxy` because it is lightweight and allows gateway credentials to be embedded in the proxy configuration pushing the burden of gateway credentials to the proxy and not the container or application of the container.

Auditing

Beyond security reasons, many regulations such as HIPAA require auditing for compliance. Our approach is two fold. We use the

ALB logging capabilities to track access to application containers and authentication. We use a logging agent to track potential privilege escalation or other security concerns on the underlying EC2 host.

The ALB provides logging [Ser20a] which will log all access to the application containers to an S3 bucket. Because in our architecture all authentication is performed using the ALB all authentication attempts both successful and more importantly failures are also logged to the bucket. Many third party log management tools are configurable to digest logs stored in this manner including Loggly, Splunk, Sumo Logic.

Another good practice is to set the target S3 bucket in a separate AWS account and only grant privileges to the logging account to write to the bucket but not delete. This ensure that even if a container or the EC2 instance is compromised, the logs can not be tampered with.

To supplement the auditing and monitoring capability one or more logging agents are installed on the EC2 instance. Essentially, this agent transmits logs of interest such as the system log `syslog` to an external log management system. Through this mechanism behaviours such as privilege escalation (e.g. `sudo`) are tracked. We use both the native AWS logging agent and a third party logging agent.

With both mechanisms in place, the preferred log management system can be configured to provide alarms when severe incidents occurs and generate reports of incidents as may be required by compliance requirements.

Other Useful Agents

Building a custom AMI image to spin up an EC2 instance to support our ECS cluster affords the opportunity to install additional agents to meet security, compliance and maintenance needs. Our best practices is to the include the following additional agents in the AMI. Some of agents are provided by AWS while some are third party.

ECS Agent: The AWS ECS agent is required in order for the EC2 instance to serve ECS containers. However, periodically updating the ECS agent is important in that potential vulnerabilities may be fixed and newer agents offer more features to aid in maintenance. Furthermore, proper configuration of features can aid in security as well. For example, the ECS agent can be configure so that the maximum lifetime of an EC2 instance is set. This is particularly useful if the AMIs for the EC2 instances are constantly being updated with security patches etc. The limited lifetime guarantees that the EC2 instances running will not be based on an AMI that is too out of date.

Systems Manager Agent: Another useful AWS Agent that can be employed is the AWS Systems Manager Agent (SSM) [Ser20e]. The SSM agent allows the "Systems Manager to update, manage and configure" the EC2 instances. This agent makes it easier to maintain EC2 instances in a centralized manner. Once again keeping an EC2 instance up to date helps reduce vulnerabilities on the node.

Anti-virus: An antivirus or antimalware agent is also recommended. The antivirus should be one that is container aware and that the container awareness feature should be active. This would facilitate pinpointing the specific container that may be compromised. Container systems such as docker are not complete virtualizations. Processes that run in a container run as processes in the native host, as such an antivirus agent inside can monitor processes that occur "inside a container". Container aware antivirus agents makes mitigation in a container environment easier. In our particular configuration, we use Sophos as the antivirus but you may have your own preferences.

Intrusion Detection: Another useful agent to be deployed on the EC2 instance is an intrusion detection agent. Like this antivirus agent, an intrusion detection agent that has container awareness capabilities is desirable and should have the capability activated. The intrusion detection agent looks for activities that are anomalous and when high risk activity is detected, it will gather as much information around the incident as it can. We use ThreatStack for our intrusion detection.

Conclusion

Presented here is a secure, collaborative infrastructure for deploying a cloud computation resources vital to our organization for scientific analysis of health related data on the Jupyter platform. The primary purpose of our infrastructure is to provide Jupyter in this environment as well as other tools such as RStudio. Our Data Science and infrastructure team is small so building a compliant infrastructure that requires little maintenance is paramount. Equally important is to safeguard against opening vulnerabilities due to misconfigurations. By following the suggestions presented here, misconfigurations err on the side of loss of functionality rather than introducing vulnerabilities.

The architecture presented here was successful in a recently performed penetration test. We hired a third party company that specializes in penetration testing and gave them normal user rights to a Jupyter notebook container and challenged them to escaped the container. The penetration testers was unable to escape the container to other parts of the system or escalate privileges to gain additional access to resources.

While the recommendations and architecture shown here rely heavily on AWS resources. No doubt elements and counterparts can be found in other cloud services such as Google Cloud and Microsoft Azure.

Snippets of code, Dockerfile, commands and other resources presented here and the corresponding poster are available at West Health's github repository at https://github.com/WestHealth/scipy2020/tree/master/cloud_infrastructure.

References

[Con19] ConSol Misc GmbH. Docker container images with "headless" vnc session, 2019. URL: https://github.com/ConSol/docker-headless-vnc-container.

[Cos18] Ciro S. Costa. Blocking ec2 metadata service from docker containers in aws, Aug 2018. URL: https://ops.tips/blog/blocking-docker-containers-from-ec2-metadata/.

[Mar18] Andrew Martin. 11 ways (not) to get hacked, 2018. URL: https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/.

[noV20] noVNC. novnc, 2020. URL: https://novnc.com/info.html.

[Pro20] Project Jupyter. Zero to jupyterhub with kubernetes, 2020. URL: https://zero-to-jupyterhub.readthedocs.io/en/latest/.

[Ser20a] Amazon Web Services. Access logs for your application load balancer, 2020. URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-access-logs.html.

[Ser20b] Amazon Web Services. Amazon ecs-optimized amis, 2020. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html.

[Ser20c] Amazon Web Services. Data encryption in efs, 2020. URL: https://docs.aws.amazon.com/efs/latest/ug/encryption.html.

[Ser20d] Amazon Web Services. Permissions boundaries for iam entities, 2020. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_boundaries.html.

[Ser20e] Amazon Web Services. Working with ssm agent, 2020. URL: https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent.html.

[Tig20] TigerVNC. Tigervnc, 2020. URL: https://tigervnc.org/.

Quasi-orthonormal Encoding for Machine Learning Applications

Haw-minn Lu[‡*]

Abstract—Most machine learning models, especially artificial neural networks, require numerical, not categorical data. We briefly describe the advantages and disadvantages of common encoding schemes. For example, one-hot encoding is commonly used for attributes with a few unrelated categories and word embeddings for attributes with many related categories (e.g., words). Neither is suitable for encoding attributes with many unrelated categories, such as diagnosis codes in healthcare applications. Application of one-hot encoding for diagnosis codes, for example, can result in extremely high dimensionality with low sample size problems or artificially induce machine learning artifacts, not to mention the explosion of computing resources needed. Quasi-orthonormal encoding (QOE) fills the gap. We briefly show how QOE compares to one-hot encoding. We provide example code of how to implement QOE using popular ML libraries such as Tensorflow and PyTorch and a demonstration of QOE to MNIST handwriting samples.

Index Terms—machine learning, classification, categorical encoding

Introduction

While most popular machine learning methods such as deep learning require numerical data as input, categorical data is very common. For example, a person's vitals could be a combination of both, they could include height, weight (numerical) and gender, race (categorical). The challenge is to convert the categorical data into a vector of some sort.

One-hot encoding which is discussed in the next section is very commonly used in machine learning but has the drawback that it can increase the dimensionality of the data by the cardinality of the category. For small category, this is not a significant issue but when categories with high cardinality are present, many problems can arise as described below.

Quasiorthonormal encoding (QOE) is a generalization of the one-hot encoding and exploits the fact that in high dimensional vector spaces, two random vectors are almost always orthogonal. The concept originated with Kůrková and Kainen [KK96]. In many ways, QOE functions the same as one-hot encoding but does not increase the dimensionality of the data to the same degree as one-hot encoding. Historically, QOE was considered for a method of encoding words but modern techniques such as *word embeddings* are now considered the state of the art method for encoding language.

* *Corresponding author: hlu@westhealth.org*
‡ *Gary and Mary West Health Institute*

Some advantages to QOE include a reduction of dimensionality over that of using one-hot encoding thus limiting effects of the "curse of dimensionality"[1] or the problem of high dimension low sample size (HDLSS). The advantage over other encodings such as binary, hash, etc. is that it does not induce artificial geometric relationships that can cause downstream bias in the results because each label in a category remains mathematically near orthogonal to the other labels.

We will briefly survey *classic* encoding methods, discuss the theoretical aspects of QOE, and present a detailed example implementation of QOE in tensorflow.

Background

Coding methods can be categorized as *classic*, *contrast*, *Bayesian* and *word embeddings*. Classic, contrast and Bayseian encoding are given a good overview treatment by Hale's blog [Hal18] with examples to be found as part of the `scikit-learn` category encoding package [McG16]. Both contrast encoding and Bayesian encoding use the statistics of the data to facilitate encoding. These two categories may be of use when more statistical analysis is required, however there has not been widespread adoption of these encoding techniques for machine learning.

Word embeddings are their own special category. [GK19]. Word embeddings are used to represent words, phrases or even entire documents as a vector so that similar meanings/concepts are mapped to vectors that are close in the target vector space. Additionally, it is adapted for encoding a large categorical features (i.e., words) into a relatively lower dimensional space.

The remainder of the section will describe some common classic categorical encodings

Ordinal Encoding

To begin our overview of fundamental encoding methods, we start with Ordinal (Label) Encoding. Ordinal encoding is the simplest and perhaps most naive approach encoding for a categorical feature --- one simply assigns a number to each member of a category. This is often how data from surveys are encoded into spreadsheets for easy storage and calculation of basic statistics. An associated data dictionary is used to convert the values back and forth between a number and a category. Take for example the case of gender, male could be encoded as 1 and female as 2, with a data dictionary as follows: `{'male': 1, 'female': 2}`

1. Mukhtar [Muk19] gives a good explanation of the curse of dimensionality as applied to data science.

Make	Ordinal	One-Hot
Toyota	1	(1,0,0,0,0)
Honda	2	(0,1,0,0,0)
Subaru	3	(0,0,1,0,0)
Nissan	4	(0,0,0,1,0)
Mitsubishi	5	(0,0,0,0,1)

TABLE 1: *Examples of Ordinal and One-Hot Encodings*

Make	Ordinal	as Binary	Binary Code
Toyota	1	001	(0,0,1)
Honda	2	010	(0,1,0)
Subaru	3	011	(0,1,1)
Nissan	4	100	(1,0,0)
Mitsubishi	5	101	(1,0,1)

TABLE 2: *Example of Binary Codes*

Suppose we have three categories of ethnic groups: White, Black, and Asian. Under ordinal encoding, suppose White is encoded as 1, Black is encoded as 2 and Asian is encoded as 3. If a machine learning classification is somehow confused between Asian and White and decides to split the difference and report the in-between value (2) which encodes Black. The issue is that arbitrary gradation between 1 and 3 introduces a natural interpolation (2) that may be nonsense. Thus, the natural ordering of the numbers imposes an ordered geometrical relationship between the categories that does not apply.

Nonetheless there are situations where ordinal encoding makes sense. For example, a 'rate your satisfaction' survey typically encodes five levels (1) terrible, (2) acceptable (3) mediocre, (4) good, (5) excellent.

One Hot Encoding

This is the most common encoding used in machine learning. One hot encoding takes a category with cardinality N and encodes each categorical value with an N-dimensional vector with a single '1' and the remainder '0's. Take as an example encoding five makes of Japanese Cars: Toyota, Honda, Subaru, Nissan, Mitsubishi. Table 1 shows a comparison of coding between ordinal and one-hot encodings.

The advantage is that one hot encoding does not induce an implicit ordering or between categories. The primary disadvantage is that the dimensionality of the problem has increased with corresponding increases in complexity, computation and "the curse of high dimensionality". This easily leads to the high dimensionality low sample size (HDLSS) situation, which is a problem for most machine learning methods.

Binary Encoding, Hash Encoding, BaseN Encoding

Somewhere in between these two are *binary encoding*, *hash encoding*, and *baseN* encoding. Binary encoding simply labels each category with a unique binary code and converts the binary code to a vector. Using the previous example of the Japanese car makes, table 2 shows an example of binary encoding.

Hash encoding assigns each category an ordinal value that is then converted into a binary hash value that is encoded as an n-tuple in the same fashion as the binary encoding. You can view hash encoding as binary encoding applied to the hashed ordinal value. Hash encoding has several advantages. First, it is open ended so new categories can be added later. Second, the resultant dimensionality can be much lower than one-hot encoding. The chief disadvantage is that categories can collide if two categories accidentally map into the same hash value. This is a *hash collision* and must be fixed separately using a resolution mechanism. Bernardi's blog [Ber18] provides a good treatment of hash coding.

Make	Ordinal	as Ternary	Ternary Code	Balanced Ternary Code
Toyota	1	01	(0,1)	(0,1)
Honda	2	02	(0,2)	(0,-1)
Subaru	3	10	(1,0)	(1,0)
Nissan	4	11	(1,1)	(1,1)
Mitsubishi	5	12	(1,2)	(1,-1)

TABLE 3: *Example of Ternary Codes*

Finally, baseN encoding is a generalization of binary encoding that uses a number base other than 2 (binary). Table 3 is an example of the Japanese car makes using base 3.

A disadvantage of all three of these techniques is that while it does reduce the dimension of the encoded feature, artificial geometric relationships may creep in between unrelated categories. For example, `(0.7,0.7)` may be confusion between Toyota and Honda or a weak Subaru result, although the effect is not as pronounced as ordinal encoding.

Decoding

Of course, with categorical encoding, the ability to decode an encoded vector back to a category can be very important. If the categorical variable is only an input to a machine learning system, retrieving a category may not be very important. For example, one may have a product rating model which delivers a rating based on a number of variables, some numeric like price, but others might be categorical like color, but since the output does not require category decoding, it is not important.

In an application such as categorization or imputation [GW18], retrieving the category from a vector is crucial. In training a modern classification model, a categorical output is often subject to an activation function which converts a vector into a probability of each category such as a *softmax* function. Essentially, the softmax is a continuous and differentiable version of a "hard max" function which would assign a `1` to the vector representing the most likely category and a `0` to all the other categories. The conversion to a probability distribution allows the use of a negative log likelihood loss function rather than the standard root mean squared error.

Typically, other classic encoding methods use thresholds to rectify a vector first into a binary or n-ary value then decode the vector back to a label in accordance to the encoding. This makes these values difficult to use as outputs of machine learning systems such as neural networks that rely on gradients due to lack of differentiability. Also, the decoding process is difficult to convert to a probability distribution, making negative log-likelihood or crossentropy loss functions more difficult to use.

Theory

In this section, we will briefly define and discuss quasiorthogonality, show how it relates to one-hot encoding and describe how this relationship can be used to develop a categorical encoding with lower cardinality.

Quasiorthogonality

In a suitably high dimensional space, two randomly selected vectors are very likely to be nearly orthogonal or quasiorthogonal. In such an n-dimensional vector space, there are sets of K vectors which are mutually quasiorthogonal where $K \gg n$. A more formal definition can be stated as follows. Given an ε, two vectors $\mathbf{x}$ and $\mathbf{y}$ are said to be *quasiorthogonal* if $\frac{|\mathbf{x}\cdot\mathbf{y}|}{\|\mathbf{x}\|\|\mathbf{y}\|} < \varepsilon$. This extends the orthogonality principle by allowing the inner product to not exactly equal zero. As an extension, we can define a quasiorthonormal *basis* by a set of normal vectors $\{\mathbf{q}_i\}$ for $i = 1,\ldots,K$ such that $|\mathbf{q}_i \cdot \mathbf{q}_j| < \varepsilon$ and $||\mathbf{q}_i|| = 1$, for all $i, j \in \{1,\ldots,K\}$, where in principle for large enough n, $K \gg n$.

The question of how large a quasiorthonormal basis can be found for a given n-dimensional vector space and ε is answered in part by the mathematical literature. [KK20] derived a lower bound for K as a function of ε and n. Namely,

$$K \geq e^{n\varepsilon^2}.$$

This means that given an ε, the size of potential quasiorthonormal basis grows at least exponentially as n grows.

One Hot Encoding Revisited

Exploiting quasiorthogonality in categorical encoding is analysis to using orthonormal basis in one-hot encoding. In a typical machine learning scenario, one hot encoding maps a variable with n categories into a set of unit vectors in a n-dimensional space: $L = \{l_i\}$ for $i = 1\ldots n$, then the one hot encoding $E_L : L \mapsto \mathbb{R}^n$ given by $l_i \mapsto \mathbf{u}_i$ where $\mathbf{u}_i$ is an orthonormal basis in $\mathbb{R}^n$. The simplest basis used is $\mathbf{u}_i = (0,0,\ldots,1,0,\ldots,0)$ where the 1 is in the ith position which is know as the *standard basis* for $\mathbb{R}^n$.

Mapping of a vector back to the original category uses the *argmax* function, so for a vector $\mathbf{z}$, $\text{argmax}(\mathbf{z}) = i$ where $z_i > z_j$ for all $j \neq i$ and the vector $\mathbf{z}$ decodes to $l_{\text{argmax}(\mathbf{z})}$. Of course, the argmax function is not easily differentiable which presents problems in ML learning algorithms that require derivatives. To fix this, a *softer* version is used called the *softargmax* or now as simply *softmax* and is defined as follows:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{1}$$

for $i = 1,2,\ldots,n$ and $\mathbf{z} = (z_1, z_2, \ldots, z_n) \in \mathbb{R}^n$ where $\mathbf{z}$ is the vector being decoded. The softmax function decodes a one-hot encoded vector into a probability density function which enables application of negative log likelihood loss functions or cross entropy losses.

Though one-hot encoding uses unit vectors with one `1` in the vector hence a *hot* component. The formalization of the one hot encoding above allows *any* orthonormal basis to be used. So to use a generalized one-hot encoding with orthonormal basis $\mathbf{u}_i$, one would map the label j to $\mathbf{u}_j$ for encoding where the $\mathbf{u}_i$ no longer have to take the standard basis form. To decode an encoded value in this framework, we would take

$$i = \text{argmax}(\mathbf{z}\cdot\mathbf{u}_1, \mathbf{z}\cdot\mathbf{u}_2, \ldots, \mathbf{z}\cdot\mathbf{u}_n). \tag{2}$$

Make	Ordinal	One-Hot	QOE
Toyota	1	$\mathbf{u}_1$	$\mathbf{q}_1$
Honda	2	$\mathbf{u}_2$	$\mathbf{q}_2$
Subaru	3	$\mathbf{u}_3$	$\mathbf{q}_3$
Nissan	4	$\mathbf{u}_4$	$\mathbf{q}_4$
Mitsubishi	5	$\mathbf{u}_5$	$\mathbf{q}_5$

TABLE 4: *Example of Quasiorthonormal Encoding*

This reduces to $\text{argmax}(\mathbf{z})$ for the standard basis. Thus, the softmax function can be expressed as the following,

$$\text{softmax}(\mathbf{z})_i = \frac{e^{\mathbf{z}\cdot\mathbf{u}_i}}{\sum_{j=1}^{n} e^{\mathbf{z}\cdot\mathbf{u}_j}}. \tag{3}$$

Encoding

The principle behind QOE is simple. A quasiorthonormal basis $\{\mathbf{q}_i\}$ is substituted for the orthonormal basis $\{\mathbf{u}_i\}$ described above. So given a quasiorthonormal basis, we can define a QOE for a set $L = \{l_i\}$ by $l_i \mapsto \mathbf{q}_i$.

Decoding $\mathbf{z}$ under QOE would use a *qargmax* function analogous to the argmax function for one-hot encoding as shown in equation 4, which is nearly identical to equation 2.

$$i = \text{argmax}(\mathbf{z}\cdot\mathbf{q}_1, \mathbf{z}\cdot\mathbf{q}_2, \ldots, \mathbf{z}\cdot\mathbf{q}_n) \tag{4}$$

Analogous to the softmax function shown of equation 3, is a *qsoftmax* function which can be expressed as

$$\text{qsoftmax}(\mathbf{z})_i = \frac{e^{\mathbf{z}\cdot\mathbf{q}_i}}{\sum_{j=1}^{K} e^{\mathbf{z}\cdot\mathbf{q}_j}} \tag{5}$$

The only real difference in the formulation is that while still operating in $\mathbb{R}^n$ we are encoding $K > n$ labels.

Returning to our example of Japanese car makes, table 4 shows one-hot encoding and QOE of the five manufacturers. In the table, encodings are represented simply as vectors where $\mathbf{u}_i$ are unit vectors in $\mathbb{R}^5$ and $\mathbf{q}_i$ are a set of quasiorthonormal vectors in $\mathbb{R}^3$. It can be shown that such a quasiorthonormal can be found in [SHS20] with the minimum mutual angle of 66°. In short, the difference between one-hot encoding and QOE is that the one-hot requires 5 dimensions and in this case QOE requires only 3.

Implementation

Mathematical

While equations 4 and 5 describe precisely mathematically how to implement decoding and activation functions. A literal implementation would not exploit the modern vectorized and accelerated computation available in such packages as `numpy`, `tensorflow` [AAB+15] and `pytorch`.

To better exploit built-in functions of these packages, we define the following $n \times K$ *change of coordinates* matrix

$$\mathbf{Q} = \begin{bmatrix} | & | & & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_K \\ | & | & & | \end{bmatrix}.$$

This transformation makes it easier to convert a set of parallel operations into matrix operations for which these aforementioned

computational packages are well suited. Mathematically, the transformation maps the representation of a category encoded by QOE to a vector representing one hot encoding. Understanding this transformation makes it simple to express `argmax` or `softmax` function's quasiorthoromal variant by equations 6 and 7, respectively.

$$\text{qargmax}(\mathbf{z}) = \text{argmax}(\mathbf{Qz}) \tag{6}$$

and

$$\text{qsoftmax}(\mathbf{z}) = \text{softmax}(\mathbf{Qz}). \tag{7}$$

The `tensorflow` and `pytorch` packages both supply optimized `softmax` functions as does `scipy` when using `numpy` arrays, making implementation of QOE not only easy, but efficient. Not only will using native functions accelerated performance, it can exploit features such as auto differentiation built into the native functions --- a useful property when using the qsoftmax function as an activation function.

Since the matrix manipulation operations and input/output shape definitions differ from package to package, we provide a qsoftmax implementation in several popular packages. In order to facilitate the most general format possible, in our examples, we will express the quasiorthogonal basis as a list of list, but the input and the output is expressed in the appropriate native class (e.g. `numpy.ndarray` in `numpy`).

Numpy

For `numpy`, the implementation is straight-forward and follows equation 7 almost literally and is given below.

```
def qsoftmax(x, basis):
    qx = np.matmul(np.asarray(basis),x)
    return softmax(qx)
```

Since `qsoftmax` given above requires the basis as a parameter as well as the input vector, it is a parameterized activation function. In many packages, only unparameterized functions can be used. The following function factory or metafunction can be used to return a `qsoftmax` function for a given basis, rather than encoding the function above in a `lambda` expression.

```
def qsoftmax(basis):
    def func(x):
        qx = np.matmul(np.asarray(basis),x)
        return softmax(qx)
    return func
```

The `softmax` function used above can be found in `scipy.special.softmax` or can easily be written as

```
def softmax(x):
     ex=np.exp(x)
     return ex/np.sum(ex)
```

Tensorflow

For `tensorflow`, the following segment of code is an implementation of the `qsoftmax` functions. By using native `tensorflow` functions, the resultant `qsoftmax` function will be automatically differentiated in a backwards neural network pass. It is also worth noting that quite often due to the way `tensorflow` performs batch processing, the input to the activation function is not a vector but an array of vectors as a `Tensor` class.

```
def qsoftmax(x, basis):
    qx = tf.matmul(tf.constant(basis), x,
                   transpose_b=True)
    return tf.nn.softmax(tf.transpose(qx))
```

A metafunction version of `qsoftmax` is also presented as this is used below in our example of MNIST handwriting classification employing QOE.

```
def qsoftmax(basis):
    def func(x):
        qx = tf.matmul(tf.constant(basis), x,
                       transpose_b=True)
        return tf.nn.softmax(tf.transpose(qx))
    return func
```

Pytorch

Presented below is a version of the `qsoftmax` function implemented using `pytorch` primitives. The use of the `squeeze` and `unsqueeze` operations convert between a 1-dimensional vector and a 2-dimension matrix having one column. This function is only designed to accept vector inputs. In some models, especially image related models, outputs of some layers may be multidimensional arrays. If your use case requires a multidimensional input to the `qsoftmax` function the code may need alteration.

```
def qsoftmax(x, basis):
   qx = torch.mm(torch.tensor(basis),
                 x.unsqueeze(0).t()).t().squeeze()
   return torch.nn.functional.softmax(qx,dim=0)
```

Construction of an Quasiorthonormal set

It is difficult to find explicit constructions of quasiorthonormal sets in the literature. Several methods are mentioned by Kainen [Kai92], but these constructions are theoretical and hard to follow. There are a number of combinatorial problems related to such as spherical codes [Eri20] and Steiner Triple Systems [LR17], which strive to find optimal solutions. These are extremely complicated mathematical constructions and not every optimal solution has been found.

Since in a high dimensional space, two random vectors are likely to be quasiorthogonal, it is tempting to take a brute force approach and simply randomly select k vectors at random and test the set. This approach is reasonable for small dimensions or small k. However, the set must have every vector be mutually quasiorthogonal and combinatoric complications quickly set in.

Suppose, the probability of any two vectors being quasiorthogonal to a given ε is p. Since there are $\binom{k}{2}$ pairs of vectors, the probability that you have a quasiorthogonal set is $p^{\binom{k}{2}}$. To put in concrete terms, if two random vectors have a 99% chance of being quasiorthogonal. Picking a set of 20 is only 14% and 30 is around 1%. Other factors conspire to make this difficult including the increasing computational complexity and the geometric differences between a cube and sphere become more pronounced as k and N grow.

As a practical matter, optimal solutions are not necessary as long as the desired characteristics of the quasiorthonormal basis are obtained. As an example, while an optimal solution finds 28 quasiorthonormal vectors with dot products of 0.5 or under are possible in seven dimensions, you may only need 10 vectors. In other words, a suboptimal solution may yield fewer vectors that are possible for a given dimension, or a larger dimension may be required to obtain the desired number of vectors that is theoretically needed.

One practical way to construct a quasiorthonormal basis is to use spherical codes which has been studied in greater detail. Spherical codes try to find a set of points on the n-dimensional hypersphere such that the minimum distance between two points

is maximized. In most constructions of spherical codes, a given point's antipodal point is also in that code set. So in order to get a quasiorthogonal set, for each pair of antipodal points, only one element of the pair is selected. Perhaps to better understand the relationship, between quasiorthonormal basis and spherical codes is that a set of spherical codes can be constructed by taking every vector in a quasiorthonormal basis and add its antipodal point.

The area of algorithmically finding a quasiorthonormal basis is scant as is finding suboptimal spherical codes. However, one such method was investigated by Gautam and Vaintrob [GV12]. Perhaps the easiest way to obtain a quasiorthonormal basis is to use spherical codes as described above but obtain the spherical code from the vast compliation of sphere codes by Sloane [SHS20].

Simple Example and Comparison

To demonstrate how QOE can be used in machine learning, we provide a simple experiment/demonstration. This demonstration in addition to showing how to construct a classification system using QOE gives an sense of the effect of QOE on accuracy. As an initial experiment, we applied QOE to classification of the Modified National Institute of Standards and Technology (MNIST) handwriting dataset [LC10], using the 60000 training examples with 10000 test examples. As there are 10 categories, we needed sets of quasiorthonormal bases with 10 elements. We took the spherical code for 24 points in 4-dimensions, giving us 12 quasi-orthogonal vectors. The maximum pairwise dot product was 0.5 leading to an angle of 60°. We also took the spherical code for 56 points in 7-dimensions, giving 28 quasi-orthogonal vectors. The maximum pairwise dot product was .33 leading to an angle of a little over 70°.

We used a hidden layer with 64 units with a ReLU activation function. Next there is a 20% dropout layer to mitigate overtraining, then an output layer whose width depends on the encoding used. We selected for this demonstration to use one of the simplest models hence there are no convolutional or pooling layers used as often seen in other sample MNIST handwriting classifiers. The following example is implemented using `tensorflow` and `keras`.

Validating the QSoftmax Function

We begin by validating the `qsoftmax` function as provided above. This is done by first constructing a reference model built on `tensorflow` and `keras` in the standard way. In fact this example is nearly identical to the presented in the *Quickstart for Beginners* guide [Ten19] for `tensorflow` with the exception that we employ a separate `Activation` for clarity.

```
normal_model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
  tf.keras.layers.Activation(tf.nn.softmax)
])
```

To validate that the `qsoftmax` function and the use of a `Lambda` layer is properly used, the `qsoftmax` metafunction is used with the identity matrix to represent the basis. Mathematically, the resultant `qsoftmax` function in the `Lambda` layer is exactly the `softmax` function. The code is shown below:

```
sanity_model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
  tf.keras.layers.Lambda(qsoftmax(numpy.identity(10,
                                  dtype=numpy.float32)))
])
```

This should function identically as the reference model because it tests that the qsoftmax function operates as expected (which it does in this case). This is useful for troubleshooting if you have difficulty.

Number of Epochs	One Hot Encoding	7-Dimensional QOE	4-Dimensional QOE
10	97.53% (97.30%)	97.24% (96.94%)	95.65% (95.15%)
20	97.68% (98.02%)	97.49% (97.75%)	95.94% (96.15%)

TABLE 5: *Results of MNIST QOE Experiment*

Examples on Quasiorthonormal Basis

To recap, for the two QOE experiments we take a set of 10 mutually quasiorthonormal vectors from a four dimensional space, and from a seven dimensional space all derived from spherical codes from tables mentioned above, and only took 10 vectors. For the code, the basis for each experiment are labeled `basis4` and `basis7`, respectively. This leads to the following models, `basis4_model` and `basis7_model`.

```
basis4_model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(4),
  tf.keras.layers.Lambda(qsoftmax(basis4))
])
basis7_model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(7),
  tf.keras.layers.Lambda(qsoftmax(basis7))
])
```

Table 5 shows the mean of the accuracy over three training runs of the validation data with training data in parentheses.

From these results, it is clear that there is some degradation in performance as the number of dimensions is reduced, but clearly QOE can be used leading to a tradeoff between accuracy and resource reduction from the reduction of dimensionality.

Extending to Spherical Encodings

A Deeper Look at Softmax

In principle, to recover a category from a potentially noisy encoded vector, the dot product of the encoded vector against each basis vector in accordance with equation 2 whether the basis is orthonormal or quasiorthonormal. If one takes a deeper dive into equations 3 and 5, it is interesting to see what these functions are doing. Figure 1 shows on the left, randomly selected values in a circle of radius 6. On the right shows the vectors after the softmax function is applied. Clearly with a few stragglers, most points either move very close to either of the basis vectors $(0,1)$ or $(1,0)$. Upon a cursory sampling of the output of the last `Dense` layer prior to application of the `softmax` function, shows that each vector component averages about 5.5 so a radius of 6 approximates the what a `softmax` function might encounter.

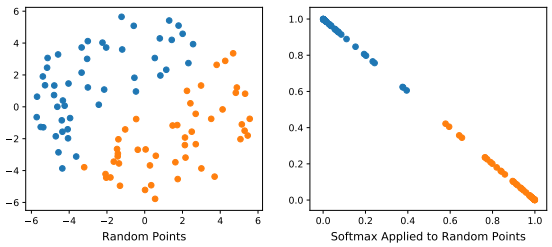


Fig. 1: *Softmax on an orthonormal basis*

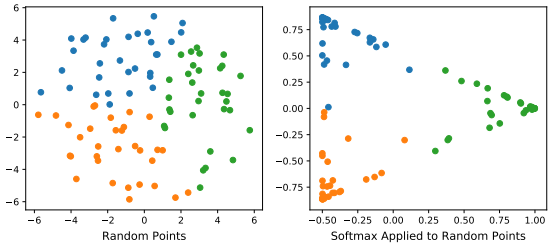


Fig. 2: *Softmax on a quasiorthogonal basis*

Similarly, figure 2 shows the same type of distribution of randomly selected values and the right shows the effect after a quasiorthonormal softmax is applied with three basis vectors. Since the qsoftmax function maps the two dimensional input into a three-dimensional space, the three-dimensional vectors are mapped back down to two dimensions using the quasiorthonormal basis. Again with the exception of a few stragglers, most points move very close to one of the three basis vectors.

Because the expectation on one-hot encoding is that the value of a given vector component be either 0 or 1 and that negative values are not expected even in a noisy environment. This is evident in figure 1, where the results are all in the first quadrant (i.e. no negative values). This raises the question could the negative values be exploited with minimal detrimental effects?

While equation 5 is intended to accept a quasiorthonormal basis, functionally there is no reason why this equation need be limited to a quasiorthonormal basis. The equation still make sense if $\{\mathbf{q}_i\}$ were replaced by any collection of normal vectors. However, the question remains as to how well that would work. So to exploit the negative regions of the coordinate system, we can see graphically what would happen if we add the antipodal vectors $(-1,0)$ and $(0,-1)$ to our standard orthonormal basis, $\{(1,0),(0,1)\}$. Applying the same type of random vector analysis to the qsoftmax function we get figure 3.

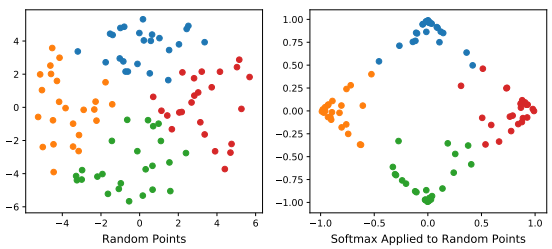


Fig. 3: *Softmax on encoded values using an orthonormal basis and antipodal points*

Make	One-Hot	Spherical Code
Toyota	(1,0,0,0,0)	(1,0,0)
Honda	(0,1,0,0,0)	(-1,0,0)
Subaru	(0,0,1,0,0)	(0,1,0)
Nissan	(0,0,0,1,0)	(0,-1,0)
Mitsubishi	(0,0,0,0,1)	(0,0,1)

TABLE 6: *Examples of Spherical Codes*

Number of Epochs	One Hot Encoding	5-Dimensional Spherical Code	3-Dimensional Spherical Code
10	97.53% (97.30%)	96.51% (96.26%)	95.37% (94.83%)
20	97.68% (98.02%)	96.82% (97.11%)	95.74% (95.83%)

TABLE 7: *Results of MNIST Spherical Coding Experiment*

So why not just use a random set of normal vectors? Despite the intuition a truly random selection will have some clustering. Geometrically, the set of normal vectors should be as evenly distributed as possible which is precisely what spherical codes are.

While it is likely that spherical codes for encoding work fine as an output such as in classification, there is an implicit relationship imposed by antipodal vector pairs especially when used as an input to a system. If you consider the spherical encoding offered in Table 6, the vector for Toyota is the negative of the vector for Honda. This is essentially telling any number system that Honda is the negative of Toyota, which may not be desirable.

With this risk in mind, we can further extend the idea to a quasiorthogonal basis by adding the antipodal vectors for each vector in the basis. The result not only doubles the number of vectors that can be used for encoding, it reduces the problem of finding a basis to that of finding spherical codes.

Spherical Codes

Spherical codes can be used in place of quasiorthonormal codes simply by allowing the $\mathbf{q}_i$ to be a collection of spherical codes not necessarily quasiorthonormal basis. Table 6 shows how the example of the five Japanese car makes could be encoded with a simple spherical code.

Since spherical codes can substitute directly into the equations for QOE, it is a simple matter to implement spherical codes $\{\mathbf{s}_i\}$ instead of quasiorthonormal basis, $\{\mathbf{q}_i\}$. As such it is a simple matter to run the same experiment on the MNIST handwriting samples as we did for QOE. First, a set of codes are defined in an `ndarray` called `code5` and `code3`. The variable `code5` consists of the standard orthonormal basis in 5 dimensions along with their antipodal unit vector to produce a set of 10 vectors in 5 dimensions. The variable `code3` is taken from [SHS20] for the 3 dimensional spherical codes with 10 vectors. Once these codes are defined, they can be substituted for `basis4` and `basis7` in the sample code above. Table 7 shows the results of the experiment with training accuracy shown in parentheses.

In this case, the 5-dimensional spherical codes performed close to the one-hot encoding by not as closely as the 7-dimension QOE codes. The 3-dimensional spherical codes performed on par with the 4-dimensional QOE codes.

While the extreme dimensionality reduction from 10 to 4 or 10 to 3 did not yield comparable performance to one-hot encoding, more modest reductions such as 10 to 7 and 10 to 5 did. It is worth considering that quasiorthogonal or spherical codes are much harder to find in low dimensions. One should note that, though we went from 10 to 7 dimensions, we did not fully exploit the space spanned by the quasiorthogonal vector set. Otherwise, we would likely have had the similar results if the categorical labels had a cardinality of 28 rather than 10.

Conclusion

These reduced dimensionality codes are not expected to improve accuracy when the training data is plentiful, but to save computation and representation by reducing the dimensionality of the coded category. As an example, in applications such as autoencoders and specifically the imputation architectures presented by [GW18] and [mLPU19], where the dimensionality not only dictates the number of outputs and inputs but also the number of hidden layers, a reduction in dimensionality has a profound impact on the size of the model used. Beyond that, the reduced dimensionality codes such as QOE and spherical codes can address problems such as the curse of dimensionality and HDLSS where for small sample sizes it may improve accuracy.

Though for the exercises presented here, the reduction of dimensionality is modest and may not seem worth the trouble. The real benefit of these codes is in extremely high cardinality situations on the order of hundreds, thousands and beyond, such as zip codes, area codes, or medical diagnostic codes.

Practically speaking, while algorithms to generate spherical codes and quasiorthonormal sets are few, [SHS20] has a vast complication of spherical codes. At the extreme end, a spherical code with 196,560 vectors is available in 24 dimensions, enough to encode nearly 100,000 labels using QOE or 200,000 labels using spherical codes, *in just 24 dimensions!*

In sum, the advantages of QOE and spherical codes are that they can reduce the dimensionality of the vector representation as compared to one-hot encoding, while not inducing artificial geometric relationships as ordinal or binary codes can. The disadvantage is that the accuracy of decoding an encoded vector in a noisy environment (such as classification output) is slightly less than one-hot encoding. This tradeoff ability makes QOE and spherical codes useful tools to be included in a data scientists toolbox alongside other established categorical coding techniques.

Experiments and code samples are made available at https://github.com/WestHealth/scipy2020/tree/master/quasiorthonormal.

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/.

[Ber18] Lucas Bernardi. Don't be tricked by the hashing trick, Jan 2018. URL: https://booking.ai/dont-be-tricked-by-the-hashing-trick-192a6aae3087.

[Eri20] Eric W. Weisstein. Spherical code, 2020. [Online; accessed 18-May-2020]. URL: https://mathworld.wolfram.com/SphericalCode.html.

[GK19] Luis Gutiérrez and Brian Keith. A systematic literature review on word embeddings. In Jezreel Mejia, Mirna Muñoz, Álvaro Rocha, Adriana Peña, and Marco Pérez-Cisneros, editors, *Trends and Applications in Software Engineering*, pages 132–141, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-01171-0_12.

[GV12] Simanta Gautam and Dmitry Vaintrob. A novel approach to the spherical codes problem. Technical report, Massachusetts Institute of Technology, 2012.

[GW18] Lovedeep Gondara and Ke Wang. Mida: Multiple imputation using denoising autoencoders. In Dinh Phung, Vincent S. Tseng, Geoffrey I. Webb, Bao Ho, Mohadeseh Ganji, and Lida Rashidi, editors, *Advances in Knowledge Discovery and Data Mining*, pages 260–272, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-93040-4_21.

[Hal18] Jeff Hale. Smarter ways to encode categorical data for machine learning: Exploring category encoders, Sep 2018. URL: https://towardsdatascience.com/smarter-ways-to-encode-categorical-data-for-machine-learning-part-1-of-3-6dca2f71b159.

[Kai92] Paul Kainen. Orthogonal dimension and tolerance. Unpublished report, Washington DC: Industrial Math, 1992.

[KK96] V. Kůrková and P. C. Kainen. A geometric method to obtain error-correcting classification by neural networks with fewer hidden units. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 2, pages 1227–1232 vol.2, 1996. doi:10.1109/ICNN.1996.549073.

[KK20] Paul C. Kainen and Věra Kůrková. *Quasiorthogonal Dimension*, pages 615–629. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-31041-7_35.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[LR17] Charles C Lindner and Christopher A Rodger. *Design theory*. CRC press, 2017. doi:10.1201/9781315107233.

[McG16] Will McGinnis. Category encoders, 2016. URL: http://contrib.scikit-learn.org/category_encoders/.

[mLPU19] Haw minn Lu, Giancarlo Perrone, and José Unpingco. Multiple imputation with denoising autoencoder using metamorphic truth and imputation feedback. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition, 16th International Conference on Machine Learning and Data Mining, MLDM 2020, Amsterdam, The Netherlands, July 20-21, 2020, Proceedings*, pages 197–208. ibai publishing, 2019. URL: http://www.ibai-publishing.org/html/proceeding2020.php.

[Muk19] Tooba Mukhtar. High dimensional data: Breaking the curse of dimensionality with python, Apr 2019. URL: https://blog.datasciencedojo.com/curse-of-dimensionality-python/.

[SHS20] N. J. A. Sloane, R. H. Hardin, and W. D. Smith. Spherical codes: Nice arrangements of points on a sphere in various dimensions, 2020. [Online; accessed 15-May-2020]. URL: http://neilsloane.com/packings/.

[Ten19] Tensorflow. Tensorflow 2 quickstart for beginners, 2019. URL: https://www.tensorflow.org/tutorials/quickstart/beginner/.

Fluctuation X-ray Scattering real-time app

Antoine Dujardin[¶], Elliott Slaugther[¶], Jeffrey Donatelli[‡§], Peter Zwart[‖§], Amedeo Perazzo[¶], Chun Hong Yoon[¶*]

https://youtu.be/IYADjGOiJhA

Abstract—The Linac Coherent Light Source (LCLS) at the SLAC National Accelerator Laboratory is an X-ray Free Electron Laser (X-FEL) facility enabling scientists to take snapshots of single macromolecules to study their structure and dynamics. A major LCLS upgrade, LCLS-II, will bring the repetition rate of the X-ray source from 120 to 1 million pulses per second and exascale High Performance Computing (HPC) capabilities will be required for the data analysis to keep up with the future data taking rates.

We present here a Python application for Fluctuation X-ray Scattering (FXS), an emerging technique for analyzing biomolecular structure from the angular correlations of FEL diffraction snapshots with one or more particles in the beam. This FXS application for experimental data analysis is being developed to run on supercomputers in near real-time while an experiment is taking place.

We discuss how we accelerated the most compute intensive parts of the application and how we used Pygion, a Python interface for the Legion task-based programming model, to parallelize and scale the application.

Index Terms—fluctuation x-ray scattering, free electron laser, real-time analysis, coherent diffractive imaging

Introduction

LCLS-II, an LCLS upgrade

The Linac Coherent Light Source (LCLS) at the SLAC National Accelerator Laboratory is an X-ray Free Electron Laser facility providing femtosecond pulses with an ultrabright beam approximately one billion times brighter than synchrotrons [WRD15]. Such a brightness allows it to work with much smaller sample sizes while the shortness allows imaging below the rotational diffusion time of the molecules and also outrunning radiation damage. With pulses of such an unprecedented brightness and shortness, scientists are able to take snapshots of single macromolecules without the need for crystallization at ambient temperature.

To push the boundaries of the science available at the light-source, LCLS is currently being upgraded after 10 years of operation. The LCLS-II upgrade will progressively increase the sampling rate from 120 pulses per second to 1 million. At these rates, the LCLS instruments will generate multiple terabytes per second of scientific data and it will therefore be critical to know what data is worth saving, requiring on-the-fly processing of the data. Earlier, users could classify and preprocess their data after the experiment, but this approach will become either prohibitive or plainly impossible. This leads us to the requirement of performing some parts of the analysis in real time during the experiment.

Quasi real time analysis of the LCLS-II datasets will require High Performance Computing, potentially at the Exascale, which cannot be offered in-house. Therefore, a pipeline to a supercomputing center is required. The Pipeline itself starts with a Data Reduction step to reduce the data size, using vetoing, feature extraction, and compression in real time. We then pass the data over the Energy Sciences Network (ESnet) to the National Energy Research Scientific Computing Center (NERSC). Currently, the ESNet connection between SLAC and NERSC is 200 Gbps capable; the plan is to upgrade this link to 400 Gbps by 2026 and to 1 Tbps by 2028. At the end of the pipeline, the actual analysis can take place on NERSC's supercomputers. This makes the whole process, from the sample to the analysis, quite challenging to change and adapt.

Moreover, LCLS experiments are typically high-risk / high-reward and involve novel setups, varying levels of requirements, and durations of only a few days. The novelty in the science can require adaptations in the algorithms, requiring the data analysis itself to be highly flexible. Furthermore, we want to give users as much freedom as possible in the way they analyze their data without expecting them to have a deep knowledge of large-scale computer programming.

Therefore, we require real time analysis, high performance computing capabilities and a complex pipeline, while requiring enough flexibility to adapt to novel experimental setups and analysis algorithms. We believe Python helps us achieve this goal given the tradeoffs involved.

FXS: an example analysis requiring HPC

While a variety of experiments can be performed at LCLS, we focus here on one specific example: Fluctuation X-ray Scattering (FXS).

X-ray scattering of particles in a solution is a common technique in the study of the structure and dynamics of macromolecules in biologically-relevant conditions and gives an understanding of their function. However, traditional methods currently used at synchrotrons suffer from the fact that the exposure time is longer than the rotation time of the particle, leading to the capture of angularly-averaged patterns. FXS techniques fully utilize the femtosecond pulses to measure diffraction patterns from multiple

¶ SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA 94025, USA
‡ Department of Applied Mathematics, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
§ Center for Advanced Mathematics for Energy Research Applications, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
‖ Molecular Biophysics and Integrated Bio-Imaging Division, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
∗ Corresponding author: yoon82@slac.stanford.edu

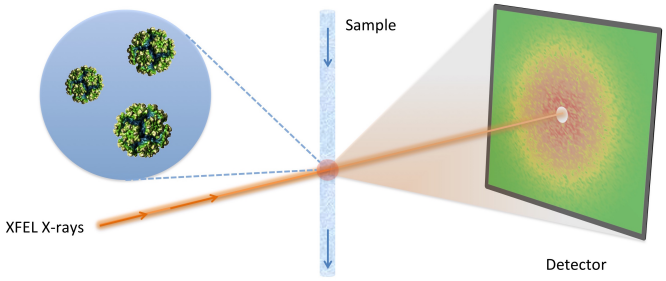


Fig. 1: *Fluctuation X-ray Scattering experiment setup.*

In an FXS experiment, femtosecond pulses from an X-ray Free Electron Laser are shot at a stream of particles in solution. The scattered light forms a diffraction pattern on the detector, aggregating the contributions of the different particles.[1]

identical macromolecules below the sample rotational diffusion times (Fig. 1). The patterns are then collected to reconstruct a 3D structure of the macromolecule or measure some of its properties. This technique was described in the late 1970s [Kam77], [KKB81] and has been widely used at LCLS [PDM+18], [KDY+17], [MLS+14], [MWQ+16].

While a few hundreds of diffraction patterns might be sufficient to reconstruct a low resolution 3-dimensional structure under ideal conditions [KDY+17], the number of snapshots required can be dramatically increased when working with low signal-to-noise ratios (e.g. small proteins) or when studying low-probability events. More interestingly, the addition of a fourth dimension, time, to study dynamical processes expands again the amount of data required. At these points, hundreds of millions or more snapshots could be required.

We present here a Python application for FXS data analysis that is being developed to run on supercomputing facilities at US Department of Energy national laboratories in near real-time while an experiment is taking place. As soon as data is produced, it is passed through a Data Reduction Pipeline on-site and sent to a supercomputer via ESNet, where reconstructions can be performed. It is critical to complete this analysis in near real-time to guide experimental decisions.

In FXS, each diffraction pattern contains several identical particles in random orientations. Information about the structure of the individual particle can be recovered by studying the two-point angular correlation of the data. To do so, the 2D images are expanded in a 3D, orientation-invariant space, where they are aggregated using the following formula:

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N}\int_0^{2\pi} I_j(q,\phi)I_j(q',\phi+\Delta\phi)d\phi \qquad (1)$$

where $I_j(q,\phi)$ represents the intensity of the j-th image, in polar coordinates. This correlator can then be used as a basis for the actual 3D reconstruction of the data (Fig. 3), using an algorithm described elsewhere [DZS15], [PDM+18].

Acceleration: getting the best out of NumPy

The expansion/aggregation step presented in Equation (1) was originally the most computation intensive part of the application, representing the vast majority of the computation time. The original implementation was processing each $I_j(q,\phi)$ image one after the other and aggregating the results. This resulted in taking 424 milliseconds per image using NumPy [Oli06], [vdWCV11] functions and a slightly better performance using Numba [LPS15]. As we illustrate in this section, rewriting this critical step allowed us to gain a factor of 40 in speed, without any other libraries or tools. The tests were performed on a node of Cori Haswell.

Let us start by simplifying Equation (1). The integral corresponds to the correlation over $I_j(q,\phi)$ and $I_j(q',\phi)$. Thanks to the convolution theorem [Arf85], we have

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N}\mathscr{F}^{-1}[\mathscr{F}[I_j(q,\phi)]\overline{\mathscr{F}[I_j(q',\phi)]}], \qquad (2)$$

where $\mathscr{F}$ represents the Fourier transform over ϕ. The inverse Fourier transform being linear, we can get it outside the sum, and on the left side. For the simplicity of the argument, we also neglect all coefficients.

Using ψ as the equivalent of ϕ in the Fourier transform and $A_j(q,\psi)$ as a shorthand for $\mathscr{F}[I_j(q,\phi)]$, we have:

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N} A_j(q,\psi)\overline{A_j(q',\psi)}. \qquad (3)$$

We end up with the naive implementation below:

```
C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    for j in range(N_RAD_BINS):
        for k in range(N_RAD_BINS):
            C2[j, k, :] += A[j] * A[k].conj()
```

taking 42.4 seconds (for 100 images), using the following parameters:

```
N_IMGS = 100
N_RAD_BINS = 300
N_PHI_BINS = 256
IMGS_SHAPE = (N_IMGS, N_RAD_BINS, N_PHI_BINS)
C2_SHAPE = (N_RAD_BINS, N_RAD_BINS, N_PHI_BINS)
```

where `N_RAD_BINS` and `N_PHI_BINS` represent the image dimensions over the q- and ϕ-axes, as well as the dataset:

```
images = np.random.random(IMGS_SHAPE)
```

We note that a typical application would be processing millions of images, but let us use 100 for the example.

This naive version can be slightly accelerated using the fact that our matrix is conjugate-symmetric:

```
C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    for j in range(N_RAD_BINS):
        C2[j, j, :] += A[j] * A[j].conj()
        for k in range(j+1, N_RAD_BINS):
            tmp = A[j] * A[k].conj()
            C2[j, k, :] += tmp
            C2[k, j, :] += tmp.conj()
```

which takes 36.0 seconds. Note that this is only 18% faster, far from a 2x speed-up.

This naive implementation should not be confused with a pure Python implementation, which is expected to be slow, since we already operate on NumPy arrays along the angular axis. Such an implementation could be approximated by:

```
A = np.fft.fft(images[i], axis=-1)
for j in range(N_RAD_BINS):
    for k in range(N_RAD_BINS):
```

1. *Copyright © P. Zwart, under the CC BY-SA 4.0 license.*

```
        for l in range(N_PHI_BINS):
            C2[j, k, l] += A[j, l] * A[k, l].conj()
```

which takes 49.1 seconds per image, i.e. about 100 times slower than the naive implementation, in accordance with the stereotype of Python being much slower than other languages for numerical computing.

A common acceleration strategy is to use Numba:

```
@numba.jit
def A_to_C2(A):
    C2 = np.zeros(C2_SHAPE, np.complex128)
    for j in range(N_RAD_BINS):
        C2[j, j, :] += A[j] * A[j].conj()
        for k in range(j+1, N_RAD_BINS):
            tmp = A[j] * A[k].conj()
            C2[j, k, :] += tmp
            C2[k, j, :] += tmp.conj()
    return C2

C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    C2 += A_to_C2(A)
```

which takes 38.5 seconds, i.e. 10% faster than the naive implementation.

When considering our problem size of up to millions of images, processing images one at a time makes sense. However, focusing on a small batch as we have been doing in these examples, a strategy can be to have NumPy and/or Numba work on arrays of images, rather than the individual images. We then have the following:

```
@numba.jit
def As_to_C2(As):
    C2 = np.zeros(C2_SHAPE, np.complex128)
    for i in range(N_IMGS):
        A = As[i]
        for j in range(N_RAD_BINS):
            C2[j, j, :] += A[j] * A[j].conj()
            for k in range(j+1, N_RAD_BINS):
                tmp = A[j] * A[k].conj()
                C2[j, k, :] += tmp
                C2[k, j, :] += tmp.conj()
    return C2

As = np.fft.fft(images, axis=-1)
C2 = As_to_C2(As)
```

which takes 11.9 seconds, i.e. 3.56 times faster. We note also here the batching of the Fast Fourier Transform.

However, such an implementation does not sound trivial using NumPy, although one can recognize a nice (generalized) Einstein sum in Equation (3), leading to:

```
As = np.fft.fft(images, axis=-1)
C2 = np.einsum('hik,hjk->ijk', As, As.conj())
```

which corresponds to expressing `C2[i, j, k]` as the sum over `h` of `As[h, i, k] * As.conj()[h, j, k]`.

This takes 17.9 seconds, which is slower than the version using Numba per batch. However, we can realize that, at this batch level, the last axis is independent from the others and that the underlying alignment of the arrays matters. Thanks to NumPy's `asfortranarray` function, however, that is not an issue. We then use the F-ordered dataset.

```
images_F = np.asfortranarray(images)
```

We observe, for the Einstein sum:

```
As = np.fft.fft(images_F, axis=-1)
C2 = np.einsum('hik,hjk->ijk', As, As.conj())
```

Implementation	Time (/100)	Speedup
Naive	42.4 s	1
Numba	38.5 s	10%
Numba, batched	11.9 s	3.56×
Einsum, F-order	4.05 s	10.5×
Dot, F-order	1.06 s	40.0×

TABLE 1: *Summary of the major time improvements.*

taking 4.05 seconds, i.e. 4.42 times faster than the C-ordered Einstein sum and 10.5 times faster than the naive implementation.

Additionally, it turns out that in our precise case, we can actually express it as a more optimized dot product:

```
As = np.fft.fft(images, axis=-1)
C2 = np.zeros(C2_SHAPE, np.complex128)
for k in range(N_PHI_BINS):
    C2[..., k] += np.dot(As[..., k].T,
                         As[..., k].conj())
```

which now brings us down to 1.37 seconds, i.e. 30.9 times faster than the naive version.

For the F-ordered case, we have:

```
As = np.fft.fft(images_F, axis=-1)
C2 = np.zeros(C2_SHAPE, np.complex128, order='F')
for k in range(N_PHI_BINS):
    C2[..., k] += np.dot(As[..., k].T,
                         As[..., k].conj())
```

taking 1.06 seconds, i.e. 29% faster than the C-ordered case and 40.0 times faster than the naive implementation. We could note that, at that speed, the main computation gets close to the time required to perform the Fast Fourier Transform, which is, in our case at least, faster on C-ordered (107 ms) than F-ordered (230 ms) data. Removing the FFT computation would yield an even starker contrast (977 ms vs. 499 ms), but would neglect the cost of the re-alignment.

In conclusion, and as summarized in Table 1, implementing this algorithm using NumPy or Numba naively gives significant improvement in computational speed compared to pure Python, but there is still a lot of room for improvement. On the other hand, such improvement does not necessarily require using fancier tools. We showed that batching our computation helped in the Numba case. From there, a batched NumPy expression looked interesting. However, it required optimizing the mathematical formulation of the problem to come up with a canonical expression, which could then be handed over to NumPy. Finally, the memory layout can have a sizable impact on the computation, while being easy to tweak in NumPy.

Parallelization: effortless scaling with Pygion

To parallelize and scale the application we use Pygion, a Python interface for the Legion task-based programming system [SA19]. In Pygion, the user decorates functions as *tasks*, and annotates task parameters with *privileges* (read, write, reduce), but otherwise need not be concerned with how tasks execute on the underlying machine. Pygion infers the dependencies between tasks based on their privileges and the values of arguments passed to tasks, and ensures that the program executes correctly, even when running on a parallel and distributed supercomputer.

To enable the distributed execution, it is necessary to separate the question of what data is needed in a given task from the

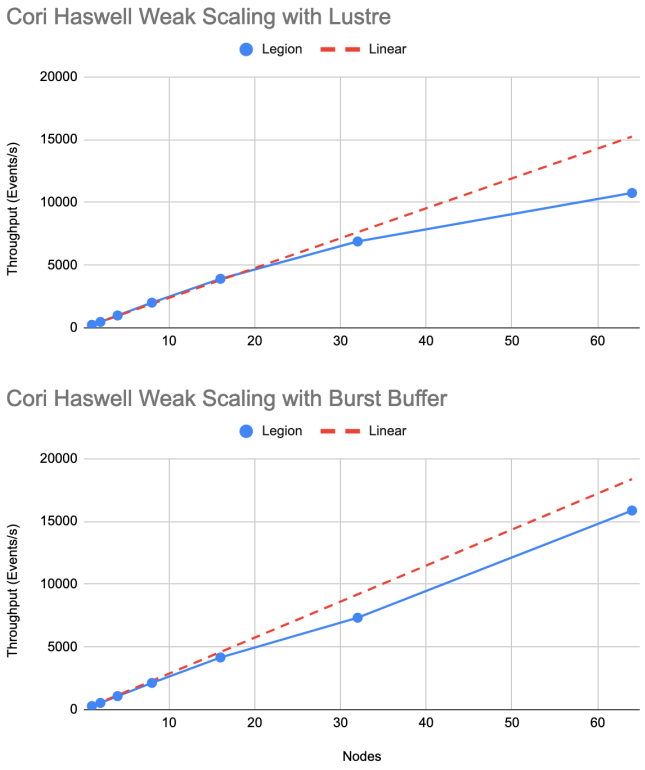


Fig. 2: *Weak scaling behavior on Cori Haswell with Lustre filesystem (top) and Burst Buffer (bottom).*

The application was run on 100,000 images per node, for up to 64 nodes on Cori Haswell. The Lustre filesystem is a high performance system running on HDDs attached to the supercomputer. The Burst Buffer corresponds to SSDs placed within the supercomputer itself used for per-job storage.

allocation of the data in a given memory or memories. This reification of the flow of data between tasks is achieved by declaring *regions*, similar to multi-dimensional Pandas dataframes [McK10]. Regions contain *fields*, each of which is similar to and exposed as a NumPy array. Regions can be partitioned into subregions, which can be processed by different tasks, allowing the parallelism. Note that regions are allocated only when needed, so it is possible (and idiomatic) to allocate a region which is larger than any single machine's memory, and then to partition into pieces that will be used by individual tasks.

We scale up to 64 Haswell nodes on NERSC's Cori supercomputer using Pygion, with 10 to 30 processes per node, to reach a throughput of more than 15,000 images per second, as illustrated in Figure 2. Compared to an equivalent MPI implementation, Pygion is easier to scale out of the box as it manages load-balancing of tasks across cores, shared memory (between distinct Python processes on a node) and provides high-level parallelization constructs. These constructs make it easy to rapidly explore different partitioning strategies, without writing or rewriting any communication code. This enabled us to quickly find a strategy that scales better than the straightforward but ultimately suboptimal strategy that we initially developed.

As an example, the most computationally intensive part of our problem is the $C_2(q,q',\Delta\phi)$ computation discussed in detail in the section above, which can trivially be parallelized over the last (angular) axis. However, the image preprocessing and the Fast Fourier Transform can only be parallelized over the first (image) axis. Given the size of the data, parallelizing between nodes would involve a lot of data movement. Parallelizing within a node, however, could help. In the MPI case, we use MPI to parallelize between nodes and within a node (MPI+MPI). If we were to introduce this optimization into such a code, one would have to create a 2-level structure such as:

```
In each node:
  Define node-level communicator
  In each rank:
    Receive and pre-process some stacks of images
  All-to-all exchange from stacks of images
    to angular sections
  In each rank:
    Process the received angular section
```

where all the data exchange has to be coded by hand.

In the Pygion case, the ability to partition the data allows us to create tasks that are unaware of the extent of the regions on which they operate. We can therefore partition these regions both over the image axis and the angular one. We end up with:

```
@task(privileges=[...])
def node_level_task(...):
    for i, batch in enumerate(data_batches):
        preprocess(input_=batch,
                   output=A_image_partition[i])
    for i in range(NUMBER_OF_PROCESSES):
        process(input_=A_angular_partition[i],
                output=C2_angular_partition[i])
```

where the data exchange is implied by the image-axis partition `A_image_partition` and the angular-axis partition `A_angular_partition` of the same region `A`.

Results

To test our framework, a dataset of 100,000 single-particle diffraction images was simulated from a lidless chaperone (mm-cpn) in its open state, using Protein Data Bank entry 3IYF [ZBS+10]. These images were processed by the algorithm described above to get the 2-point correlation function, $C_2(q,q',\Delta\phi)$, described in Equation (1). This correlation function was first filtered and reduced using the methods described in [PDM+18], and then the reconstruction algorithm in [DZS15] was applied to reconstruct the electron density of the chaperone from the reduced correlations, yielding the reconstruction shown in Figure 3.

To obtain this result, the correlation function was filtered and reduced using the Multi-Tiered Iterative Filtering (M-TIF) algorithm [PDM+18]. In particular, M-TIF uses several iterations of Tikhonov regularization, linear pseudo inversion, and principal component analysis to fit three tiers of expansions to the data: a Legendre polynomial expansion in theta, a Hankel-transformed Fourier-Bessel expansion in q and q', and a low-rank eigenvalue decomposition on the matrices of Fourier-Bessel coefficients. The number of terms needed in each expansion step is limited and determined by an upper-bound diameter estimate of the protein sample. Once these coefficients are determined, their corresponding series expansions are computed to produce a filtered correlation function, along with a reduced set of Legendre polynomial expansion coefficients on a coarse q-grid, which is used in the reconstruction (See [PDM+18] for more details on the filtering).

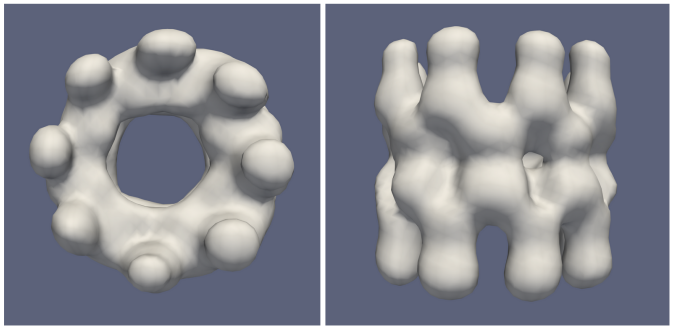


Fig. 3: *Reconstruction of a lidless chaperone (mm-cpn) in its open state from simulated diffraction patterns.*
The 2-point correlation function was computed on the simulated dataset as described in the present document. It was then filtered, reduced, and fed to a reconstruction algorithm described elsewhere [PDM+18], [DZS15] to yield the reconstruction above.

These Legendre expansion coefficients can be directly related to the protein sample. In particular, the coefficients are equal to the inner products of spherical harmonic coefficients of the 3D intensity function, which is defined as the squared magnitude of the Fourier transform of the sample's electron density [Kam77]. This relation can be expressed as two tiers of phase problems that need to be solved to reconstruct the underlying density: a hyperphase problem to recover the intensity function from the Legendre coefficients, and a classical scalar phase problem to recover the density from the intensity. In order to reconstruct the sample, we apply the Multi-Tiered Iterative Phasing (M-TIP) algorithm [DZS15] to the Legendre coefficients computed from the M-TIF filtering/reduction procedure. M-TIP works by using a set of computationally efficient projection operators in a self-consistent iteration to simultaneously solve both tiers of phase problems and reconstruct the sample from the Legendre coefficients.

After acceleration and parallelization, we now reach a throughput of about 230 images per second on a single node of Cori Haswell. This would allow us to process in real time the output of an FXS experiment at LCLS-I, which produces 120 images per second. Such a rapid processing would make possible to give scientists immediate feedback on the quality of their data. After scaling to up to 64 nodes, the throughput of about 15,000 images per second would be sufficient to follow up with the early abilities of LCLS-II, although further acceleration and scaling will be required to match the data being produced as LCLS-II increases its pulse rate dramatically over the following years.

Interestingly, one might note from Equations 1, 2, or 3 that computing the correlation function involves a sum over all the images. The output of that computation, however, no longer depends on the number of images in the dataset. The size of the correlation function $C_2(q,q',\Delta\phi)$ is, therefore, only dependent on the resolution over the q, q', and $\Delta\phi$ axes. As a consequence, the computational complexity of the post-processing of the correlation function and the reconstruction algorithm does not scale with the amount of data being processed.

Conclusion

The Linac Coherent Light Source provides scientists with the ability of X-ray diffraction patterns with much higher brightness and much shorter timescales, allowing experiments not possible elsewhere. With its upgrades LCLS-II in 2021 and LCLS-II-HE (High Energy) in 2025, LCLS experiments will produce up to millions of X-ray pulses per second and generate commensurate amounts of data. In some cases, such as the FXS technique described in this paper, the processing of the dataset will require High Performance Computing at a scale that can no longer be provided in-house.

We showed that Python gives us and our users the flexibility to adapt the analysis pipeline to new experiments. The main drawback of Python is that implementing new algorithms without relying on specialized libraries can be problematically slow. However, we illustrate with our example that spending some time optimizing the math of the problem (rather than the code) and being aware of the strengths and weaknesses of NumPy and Numba can allow us to achieve drastically better performances, without the need to develop or use external libraries.

Finally, we used Pygion to manage the parallelization of the problem, which allows us to design applications that scale much more naturally than MPI at a given level of coding effort, and in particular has allowed us to explore different parallelization strategies more rapidly, leading ultimately to a more scalable solution than what we otherwise might have been able to find.

Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Use of the Linac Coherent Light Source (LCLS), SLAC National Accelerator Laboratory, is supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences under Contract No. DE-AC02-76SF00515.

References

[Arf85] G Arfken. Convolution theorem. In *Mathematical Methods for Physicists*, chapter 15.5, pages 810–814. Academic Press, Orlando, FL, 3 edition, 1985.

[DZS15] Jeffrey J Donatelli, Peter H Zwart, and James A Sethian. Iterative phasing for fluctuation X-ray scattering. *Proceedings of the National Academy of Sciences of the United States of America*, 112(33):10286–91, 2015. doi:10.1073/pnas.1513738112.

[Kam77] Zvi Kam. Determination of Macromolecular Structure in Solution by Spatial Correlation of Scattering Fluctuations. *Macromolecules*, 10(5):927–934, 1977. doi:10.1021/ma60059a009.

[KDY+17] Ruslan P. Kurta, Jeffrey J. Donatelli, Chun Hong Yoon, Peter Berntsen, Johan Bielecki, Benedikt J. Daurer, Hasan Demirci, Petra Fromme, Max Felix Hantke, Filipe R.N.C. Maia, Anna Munke, Carl Nettelblad, Kanupriya Pande, Hemanth K.N. Reddy, Jonas A. Sellberg, Raymond G. Sierra, Martin Svenda, Gijs Van Der Schot, Ivan A. Vartanyants, Garth J. Williams, P. Lourdu Xavier, Andrew Aquila, Peter H. Zwart, and Adrian P. Mancuso. Correlations in Scattered X-Ray Laser Pulses Reveal Nanoscale Structural Features of Viruses. *Physical Review Letters*, 119(15), 2017. doi:10.1103/PhysRevLett.119.158102.

[KKB81] Z Kam, M. H.J. Koch, and J. Bordas. Fluctuation x-ray scattering from biological particles in frozen solution by using synchrotron radiation. *Proceedings of the National Academy of Sciences of the United States of America*, 78(6 I):3559–3562, 1981. doi:10.1073/pnas.78.6.3559.

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

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stefan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi:10.25080/Majora-92bf1922-00a.

[MLS+14] Derek Mendez, Thomas J. Lane, Jongmin Sung, Jonas Sellberg, Clément Levard, Herschel Watkins, Aina E. Cohen, Michael Soltis, Shirley Sutton, James Spudich, Vijay Pande, Daniel Ratner, and Sebastian Doniach. Observation of correlated X-ray scattering at atomic resolution. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 369(1647):20130315, 2014. doi:10.1098/rstb.2013.0315.

[MWQ+16] Derek Mendez, Herschel Watkins, Shenglan Qiao, Kevin S. Raines, Thomas J. Lane, Gundolf Schenk, Garrett Nelson, Ganesh Subramanian, Kensuke Tono, Yasumasa Joti, Makina Yabashi, Daniel Ratner, and Sebastian Doniach. Angular correlations of photons from solution diffraction at a free-electron laser encode molecular structure. *IUCrJ*, 3(6):420–429, 2016. doi:10.1107/S2052252516013956.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PDM+18] Kanupriya Pande, Jeffrey J Donatelli, Erik Malmerberg, Lutz Foucar, Christoph Bostedt, Ilme Schlichting, and Petrus H Zwart. Ab initio structure determination from experimental fluctuation X-ray scattering data. *Proceedings of the National Academy of Sciences of the United States of America*, 115(46):11772–11777, 2018. doi:10.1073/pnas.1812064115.

[SA19] Elliott Slaughter and Alex Aiken. Pygion: Flexible, Scalable Task-Based Parallelism with Python. In *Proceedings of PAW-ATM 2019: Parallel Applications Workshop, Alternatives to MPI+X, Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58–72. Institute of Electrical and Electronics Engineers (IEEE), 2019. doi:10.1109/PAW-ATM49560.2019.00011.

[vdWCV11] Stéfan van der Walt, Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22–30, 2011. doi:10.1109/MCSE.2011.37.

[WRD15] William E. White, Aymeric Robert, and Mike Dunne. The linac coherent light source. *Journal of Synchrotron Radiation*, 22:472–476, 2015. doi:10.1107/S1600577515005196.

[ZBS+10] Junjie Zhang, Matthew L. Baker, Gunnar F. Schröder, Nicholai R. Douglas, Stefanie Reissmann, Joanita Jakana, Matthew Dougherty, Caroline J. Fu, Michael Levitt, Steven J. Ludtke, Judith Frydman, and Wah Chiu. Mechanism of folding chamber closure in a group II chaperonin. *Nature*, 463(7279):379–383, 2010. doi:10.1038/nature08701.

HOOMD-blue version 3.0 A Modern, Extensible, Flexible, Object-Oriented API for Molecular Simulations

Brandon L. Butler[‡*], Vyas Ramasubramani[‡], Joshua A. Anderson[‡], Sharon C. Glotzer[‡§¶‖]

https://youtu.be/fIFPYZsOVqI

Abstract—HOOMD-blue is a library for running molecular dynamics and hard particle Monte Carlo simulations that uses pybind11 to provide a Python interface to fast C++ internals. The package is designed to scale from a single CPU core to thousands of NVIDIA or AMD GPUs. In developing HOOMD-blue version 3.0, we significantly improve the application protocol interface (API) by making it more flexible, extensible, and Pythonic. We have also striven to provide simpler and more performant entry points to the internal C++ classes and data structures. With these updates, we show how HOOMD-blue users will be able to write completely custom Python classes which integrate directly into the simulation run loop and analyze previously inaccessible data. Throughout this paper, we focus on how these goals have been achieved and explain design decisions through examples of the newly developed API.

Index Terms—molecular dynamics, molecular simulations, Monte Carlo simulations, object-oriented

Introduction

Molecular simulation has been an important technique for studying the equilibrium properties of molecular systems since the 1950s. The two most common methods for this purpose are molecular dynamics and Monte Carlo simulations [MRR+], [AW]. Molecular dynamics (MD) is the application of Newton's laws of motion to molecular system, while Monte Carlo (MC) methods employ a Markov chain to sample from equilibrium configurations. Since their inception these tools have been used to study numerous systems, examples include colloids [DEG], metallic glasses [FIE], and proteins [DZK+], among others.

Today many software packages exist for these purposes. LAMMPS [Pli], GROMACS [BvdSvD], [AMS+], OpenMM [ESC+], ESPResSo [WWS+], [GTK+] and Amber [SCW], [CCD+] are a few examples of popular MD packages, while Cassandra [SMM+] and MCCCS Towhee [Mar] provide MC simulation capabilities. Implementations on high performance GPUs [SMAG], parallel architectures [NBB+], and the greater accessibility of computational power have tremendously improved the length [BCR+] and time [SDS+] scales of simulations from those conducted in the mid 1900s. The flexibility and generality of such tools has dramatically increased the usage of molecular simulations, which has in turn led to demands for even more customizable software packages that can be tailored to very specific simulation requirements. Different tools have taken different approaches to enabling this, such as the text-file scripting in LAMMPS, the command line interface provided by GROMACS, and the Python, C++, C, and Fortran bindings of OpenMM. Recently, programs that have used other interfaces have also added Python bindings such as LAMMPS and GROMACS.

In the development of these tools, the requirements for the software to enable good science became more obvious. Having computational research that is Transferable, Reproducible, Usable (by others), and Extensible (TRUE) [TGM+] is necessary for fully realizing the potential of computational molecular science. HOOMD-blue is part of the MoSDeF project which seeks to bring these traits to the wider computational molecular science community through packages like mbuild [KSJ+] and foyer [KST+] which are Python packages that generalize generating initial particle configurations and force fields respectively across a variety of simulation back ends [CG], [TGM+]. This effort in increased TRUEness is one of many motivating factors for HOOMD-blue version 3.0.

HOOMD-blue [ALT], [GNA+], [AGG], an MD and MC simulations engine with a C++ back end, provides to use a Python API facilitated through pybind11 [JRM]. The package is open-source under the 3-clause BSD license, and the code is hosted on GitHub (https://github.com/glotzerlab/hoomd-blue). HOOMD-blue was initially released in 2008 as the first fully GPU-enabled MD simulation engine using NVIDIA GPUs through CUDA. Since its initial release, HOOMD-blue has remained under active development, adding numerous features over the years that have increased its range of applicability, including adding support for domain decomposition (dividing the simulation box among MPI ranks) in 2014 and recent developments that enable support for AMD in addition to NVIDIA GPUs.

Despite its great flexibility, the package's API still has certain key limitations. In particular, since its inception HOOMD-blue has been designed around some maintenance of global state. The original releases of HOOMD-blue provided Python scripting capabilities based on an imperative programming model, but it required that these scripts be run through HOOMD-blue's mod-

* Corresponding author: butlerbr@umich.edu
‡ University of Michigan, Department of Chemical Engineering
§ University of Michigan, Department of Material Science and Engineering
¶ University of Michigan, Department of Physics
‖ University of Michigan, Biointerfaces Institute

ified interpreter that was responsible for managing this global state. Version 2.0 relaxed this restriction, allowing the use of HOOMD-blue within ordinary Python scripts and introducing the `SimulationContext` object to encapsulate the global state to some degree, thereby allowing multiple largely independent simulations to coexist in a single script. However, this object remained largely opaque to the user, in many ways still behaving like a pseudo-global state, and version 2.0 otherwise made minimal modifications to the HOOMD-blue Python API, which was largely inspired by and reminiscent of the structure of other simulation software, particularly LAMMPS.

In this paper, we describe the upcoming 3.0 release of HOOMD-blue, which is a complete redesign of the API from the ground up to present a more transparent and Pythonic interface for users. Version 3.0 aspires to match the intuitive APIs provided by other Python packages like SciPy [VGO+], NumPy [vdWCV], scikit-learn [PVG+], and matplotlib [Hun], while simultaneously adding seamless interfaces by which such packages may be integrated into simulation scripts using HOOMD-blue. Global state has been completely removed, instead replaced by a highly object-oriented model that gives users explicit and complete control over all aspects of simulation configuration. Where possible, the new version also provides performant, Pythonic interfaces to data stored by the C++ back end. Over the next few sections, we will use examples of HOOMD-blue's version 3.0 API (which is still in development at the time of writing) to highlight the improved extensibility, flexibility, and ease of use of the new HOOMD-blue API.

General API Design

Rather than beginning with abstract descriptions, we will introduce the new API by example. The script below illustrates a standard MD simulation of a Lennard-Jones fluid using the version 3.0 API. Each of the elements of this script is introduced throughout the rest of this section. We also show a rendering of the particle configuration in Figure (1).

```
import hoomd
import hoomd.md
import numpy as np

device = hoomd.device.Auto()
sim = hoomd.Simulation(device)

# Place particles on simple cubic lattice.
N_per_side = 14
N = N_per_side ** 3
L = 20
xs = np.linspace(0, 0.9, N_per_side)
x, y, z = np.meshgrid(xs, xs, xs)
coords = np.array(
    (x.ravel(), y.ravel(), z.ravel())).T

# One way to define an initial system state is
# by defining a snapshot and using it to
# initialize the system state.
snap = hoomd.Snapshot()
snap.particles.N = N
snap.configuration.box = hoomd.Box.cube(L)
snap.particles.position[:] = (coords - 0.5) * L
snap.particles.types = ['A']

sim.create_state_from_snapshot(snap)

# Create integrator and forces
integrator = hoomd.md.Integrator(dt=0.005)
langevin = hoomd.md.methods.Langevin(
    hoomd.filter.All(), kT=1., seed=42)
```

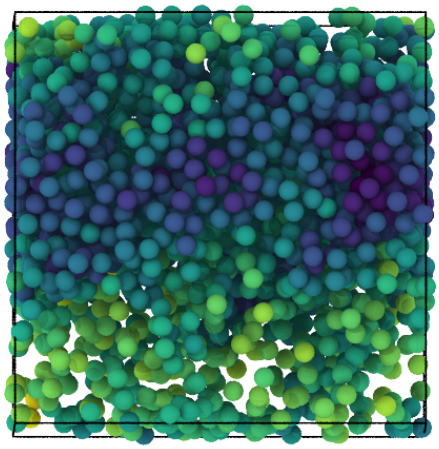


Fig. 1: *A rendering of the Lennard-Jones fluid simulation script output. Particles are colored by the Lennard-Jones potential energy that is logged using the HOOMD-blue `Logger` and `GSD` class objects. Figure is rendered in OVITO [Stu] using the Tachyon [Sto] renderer.*

```
integrator.methods.append(langevin)

nlist = hoomd.md.nlist.Cell()
lj = hoomd.md.pair.LJ(nlist, r_cut=2.5)
lj.params[('A', 'A')] = dict(
    sigma=1., epsilon=1.)
integrator.forces.append(lj)

# Set up output
gsd = hoomd.output.GSD('trajectory.gsd', trigger=100)
log = hoomd.logging.Logger()
log += lj
gsd.log = log

sim.operations.integrator = integrator
sim.operations.analyzers.append(gsd)
sim.run(100000)
```

Simulation, Device, State, Operations

Each simulation in HOOMD-blue is now controlled through three main objects which are joined together by the `Simulation` class: the `Device`, `State`, and `Operations` classes. Figure (2) shows this relationship with some core attributes/methods for each class. Each `Simulation` object holds the requisite information to run a full molecular dynamics or Monte Carlo simulation, thereby circumventing any need for global state information. The `Device` class denotes whether a simulation should be run on CPUs or GPUs and the number of cores/GPUs it should run on. In addition, the device manages custom memory tracebacks, profiler configurations, and the MPI communicator among other things.

The `State` class stores the system data (e.g. particle positions, orientations, velocities, the system box). As shown in our example, the state can be initialized from a snapshot, after which the data can be accessed and modified in two ways. One option is for users to operate on a new `Snapshot` object, which exposes NumPy arrays that store a copy of the system data. To construct a snapshot, all system data distributed across MPI ranks must be gathered and combined by the root rank. Setting the state using the snapshot API requires assigning a modified snapshot to the system state (i.e. all system data is reset upon setting). The advantages to this approach come from the ease of use of working with a

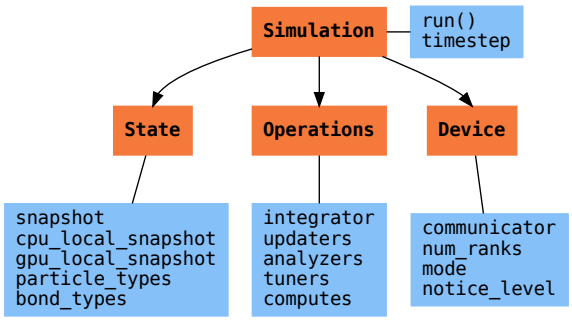


Fig. 2: *Diagram of core objects with some attributes and methods. Classes are in bold and orange; attributes and methods are blue. Figure is made using Graphviz [EGK+], [GKNV].*

single object containing the complete description of the state. The following snippet showcases how this approach can be used to set the z position of all particles to zero.

```
snap = sim.state.snapshot
# snapshot only stores data on rank 0
if snap.exists:
    # set all z positions to 0
    snap.particles.position[:, 2] = 0
sim.state.snapshot = snap
```

The other API for accessing `State` data is via a zero-copy, rank-local access to the state's data on either the GPU or CPU. On the CPU, we expose the buffers as `numpy.ndarray`-like objects through provided hooks such as `__array_ufunc__` and `__array_interface__`. Similarly, on the GPU we mock much of the CuPy [zot] `ndarray` class if it is installed; however, at present the CuPy package provides fewer hooks, so our integration is more limited. Whether or not CuPy is installed, we use version 2 of the `__cuda_array_interface__` protocol for GPU access (compatibility with our GPU buffers in Python therefore depends on the support of version 2 of this protocol). This provides support for libraries such as Numba's [LPS] GPU just-in-time compiler and PyTorch [PGM+]. We chose to mock NumPy-like interfaces rather than expose `ndarray` objects directly out of consideration for memory safety. To ensure data integrity, we restrict the data to only be accessible within a specific context manager. This approach is much faster than using the snapshot API because it uses HOOMD-blue's data buffers directly, but the nature of providing zero-copy access requires that users deal directly with the domain decomposition since only data for a MPI rank's local simulation box is stored by a given rank. The example below modifies the previous example to instead use the zero-copy API.

```
with sim.state.cpu_local_snapshot as data:
    data.particles.position[:, 2] = 0

# assumes CuPy is installed
with sim.state.gpu_local_snapshot as data:
    data.particles.position[:, 2] = 0
```

The last of the three classes, `Operations`, holds the different *operations* that will act on the simulation state. Broadly, these consist of 3 categories: updaters, which modify simulation state; analyzers, which observe system state; and tuners, which tune the hyperparameters of other operations for performance. Although updaters and analyzers existed in version 2.x (tuners are a version 3.0 split from updaters), these *operations* have undergone a significant API overhaul for version 3.0 to support one of the more far-reaching changes to HOOMD-blue: the deferred initialization model.

Operations in HOOMD-blue are generally implemented as two classes, a user-facing Python object and an internal C++ object which we denote as the *action* of the operation. On creation, these C++ objects typically require a `Device` and a C++ `State` in order to, for instance, initialize appropriately sized arrays. Unfortunately this requirement restricts the order in which objects may be created since devices and states must exist first. This restriction could create potential confusion for users who forget this ordering and would also limit the composability of modular simulation components by preventing, for instance, the creation of a simple force field without the prior existence of a `Device` and a `State`. To circumvent these difficulties, the new API has moved to a deferred initialization model in which C++ objects are not created until the corresponding Python objects are *attached* to a `Simulation`, a model we discuss in greater detail below.

Deferred C++ Initialization

The core logic for the deferred initialization model is implemented in the `_Operation` class, which is the base class for all operations in Python. This class contains the machinery for attaching/detaching operations to/from their C++ counterparts, and it defines the user interface for setting and modifying operation-specific parameters while guaranteeing that such parameters are synchronized with attached C++ objects as appropriate. Rather than handling these concerns directly, the `_Operation` class manages parameters using specially defined classes that handle the synchronization of attributes between Python and C++: the `ParameterDict` and `TypeParameterDict` classes. In addition to providing transparent `dict`-like APIs for the automatically synchronized setting of parameters, these classes also provide strict validation of input types, ensuring that user inputs are validated regardless of whether or not operations are attached to a simulation.

Each class supports validation of their keys, and they can be used to define the structure and validation of arbitrarily nested dictionaries, lists, and tuples. Likewise, both support default values, but to a varying degree due to their differing purposes. `ParameterDict` acts as a dictionary with additional validation logic. However, the `TypeParameterDict` represents a dictionary in which each entry is validated by the entire defined schema. This distinction occurs often in simulation contexts as simulations with multiple types of particles, bonds, angles, etc. must specify certain parameters for each type. In practice this distinction means that the `TypeParameterDict` class supports default specification with arbitrary nesting, while the `ParameterDict` has defaults but these are equivalent to object attribute defaults. An example `TypeParameterDict` initialization and use of both classes can be seen below.

```
# Specification of Sphere's shape TypeParameterDict
TypeParameterDict(
    diameter=float,
    ignore_statistics=False,
    orientable=False,
    len_keys=1)

from hoomd.hpmc.integrate import Sphere
```

```
sphere = Sphere(seed=42)
# Set nselect parameter using ParameterDict
sphere.nselect = 2
# Set shape for type 'A' using TypeParameterDict
sphere.shape['A'] = {'diameter': 1.}
# Set shape for types 'B', 'C', and 'D'
sphere.shape[['B', 'C', 'D']] = {'diameter': 0.5}
```

The specification defined above sets defaults for `ignore_statistics` and `orientable` (the purpose of these is outside the scope of the paper), but requires the setting of the `diameter` for each type.

To store lists of operations that must be attached to a simulation, the analogous `SyncedList` class transparently handles attaching of operations.

```
import hoomd

ops = hoomd.Operations()
gsd = hoomd.output.GSD('example.gsd')
# Append to the SyncedList ops.analyzers
ops.analyzers.append(gsd)
```

These classes also have the ancillary benefit of improving error messaging and handling. An example error message for trying to set `sigma` for *A*-*A* interactions in the Lennard-Jones pair potential to a string (i.e. `lj.params[('A', 'A')] = {'sigma': 'foo', 'epsilon': 1.}` would provide the error message,

> TypeConversionError: For types [('A', 'A')], error In key sigma: Value foo of type <class 'str'> cannot be converted using OnlyType(float). Raised error: value foo not convertible into type <class 'float'>.

Previously, the equivalent error would be "TypeError: must be real number, not str", the error would not be raised until running the simulation, and the line setting sigma would not be in the stack trace given.

Logging and Accessing Data

Logging simulation data for analysis is a critical feature of molecular simulation software packages. Up to now, HOOMD-blue has supported logging through an analyzer interface that simply accepted a list of quantities to log, where the set of valid quantities was based on what objects had been created at any point and stored to the global state. The creation of the base `_Operation` class has allowed us to simultaneously simplify and increase the flexibility of our logging infrastructure. The `Loggable` metaclass of `_Operation` allows all subclasses to expose their loggable quantities by marking Python properties or methods to query.

The actual task of logging data is accomplished by the `Logger` class, which provides an interface for logging most HOOMD-blue objects and custom user quantities. In the example script from the General API Design section above, we show that the `Logger` can add an operation's loggable quantities using the += operator. The utility of this class lies in its intermediate representation of the data. Using the HOOMD-blue namespace as the basis for distinguishing between quantities, the `Logger` maps logged quantities into a nested dictionary. For example, logging the Lennard-Jones pair potentials total energy would produce this dictionary by a `Logger` object `{'md': {'pair': {'LJ': {'energy': (-1.4, 'scalar')}}}}` where `'scalar'` is a flag to make processing the logged output easier. In real use cases, the dictionary would likely be filled with many other quantities.

Version 3.0 of HOOMD-blue uses properties extensively to expose object data such as the total potential energy of all pair potentials, the trial move acceptance rate in MC integrators, and thermodynamic variables like temperature or pressure, all of which can be used directly or stored through the logging interface. To support storing these properties, the logging is quite general and supports scalars, strings, arrays, and even generic Python objects. By separating the data collection from the writing to files, and by providing such a flexible intermediate representation, HOOMD-blue can now support a range of back ends for logging; moreover, it offers users the flexibility to define their own. For instance, while logging data to text files or standard out is supported out of the box, other back ends like MongoDB, Pandas [McK], and Python pickles can now be implemented on top of the existing logging infrastructure. Consistent with the new approach to logging, HOOMD-blue version 3.0 makes simulation output an opt-in feature even for common outputs like performance and thermodynamic quantities. In addition to this improved flexibility in storage possibilities, for HOOMD-blue version 3.0 we have exposed more of an object's data than had previously been available through adding new properties to objects. For example, pair potentials now expose *per-particle* potential energies at any given time (this data is used to color Figure (1)).

In conjunction with the deferred initialization model, the new logging infrastructure also allows us to more easily export an object's state (not to be confused with the simulation state). Due to the switch to deferred initialization, all operation state information is now stored directly in Python, so we have made object state a loggable quantity. All operations also provide a `from_state` factory method that can reconstruct the object from the state, dramatically increasing the restartability of simulations since the state of each object can be saved at the end of a given run and read at the start of the next.

```
from hoomd.hpmc.integrate import Sphere

sphere = Sphere.from_state('example.gsd', frame=-1)
```

This code block would create a `Sphere` object with the parameters stored from the last frame of the gsd file `example.gsd`.

User Customization

A major improvement in HOOMD-blue version 3 is the ease with which users can customize their simulations in previously impossible ways. The changes that enable this improvement generally come in two flavors, the generalization of existing concepts in HOOMD-blue and the introduction of a completely new `Action` class that enables the user to inject arbitrary actions into the simulation loop. In this section, we first discuss how concepts like periods and groups have been generalized from previous iterations of HOOMD-blue and then show how users can inject completely novel routines to actually modify the behavior of simulations.

Triggers

In HOOMD-blue version 2.x, everything that was not run on every timestep had a period and phase associated with it. The timesteps the operation was run on could then be determined by the expression, `timestep % period - phase == 0`. In our refactoring and development, we recognized that this concept could be made much more general and consequently more flexible. Objects do not have to be run on a periodic timescale; they just need some indication of when to run. In other words, the operations needed to be *triggered*. The `Trigger` class encapsulates this

concept, providing a uniform way of specifying when an object should run without limiting options. `Trigger` objects return a Boolean value when called with a timestep (i.e. they are functors). Each operation that requires triggering is now associated with a corresponding `Trigger` instance which informs the simulation when the operation should run. The previous behavior is now available through the `Periodic` class in the `hoomd.trigger` module. However, this approach enables much more sophisticated logic through composition of multiple triggers such as `Before` and `After` which return `True` before or after a given timestep with the `And`, `Or`, and `Not` subclasses that function as logical operators on the return value of the composed `Triggers`.

In addition to the flexibility the `Trigger` class provides by abstracting out the concept of triggering an operation, we use pybind11 to easily allow subclasses of the `Trigger` class in Python. This allows users to create their own triggers in pure Python that will execute in HOOMD-blue's C++ back end. An example of such a subclass that reimplements the functionality of HOOMD-blue version 2.x can be seen below.

```
from hoomd.trigger import Trigger

class CustomTrigger(Trigger):
    def __init__(self, period, phase=0):
        super().__init__()
        self.period = period
        self.phase = phase

    def __call__(self, timestep):
        v = timestep % self.period - self.phase == 0
        return v
```

User-defined subclasses of `Trigger` are not restricted to simple algorithms or even stateless ones; they can implement arbitrarily complex Python code as demonstrated in the Large Examples section's first code snippet.

Variants

`Variant` objects are used in HOOMD-blue to specify quantities like temperature, pressure, and box size which can vary over time. Similar to `Trigger`, we generalized our ability to linearly interpolate values across timesteps (`hoomd.variant.linear_interp` in HOOMD-blue version 2.x) to a base class `Variant` which generalizes the concept of functions in the semi-infinite domain of timesteps $t \in \mathbb{Z}_0^+$. This allows sinusoidal cycling, non-uniform ramps, and other behaviors. Like `Trigger`, `Variant` can be a direct subclass of the C++ class. An example of a sinusoidal cycling variant is shown below.

```
from math import sin
from hoomd.variant import Variant

class SinVariant(Variant):
    def __init__(self, frequency, amplitude,
                 phase=0, center=0):
        super().__init__()
        self.frequency = frequency
        self.amplitude = amplitude
        self.phase = phase
        self.center = center

    def __call__(self, timestep):
        tmp = self.frequency * timestep
        tmp = sin(tmp + self.phase)
        return self.amplitude * tmp + self.center

    def _min(self):
        return self.center - self.amplitude

    def _max(self):
        return self.center + self.amplitude
```

ParticleFilters

Unlike `Trigger` or `Variant`, `ParticleFilter` is not a generalization of an existing concept but the splitting of one class into two. However, this split is also targeted at increasing flexibility and extensibility. In HOOMD-blue version 2.x, the `ParticleGroup` class and subclasses served to provide a subset of particles within a simulation for file output, application of thermodynamic integrators, and other purposes. The class hosted both the logic for storing the subset of particles and filtering them out from the system. After the refactoring, `ParticleGroup` is only responsible for the logic to store and perform some basic operations on a set of particle tags (a means of identifying individual particles), while the new class `ParticleFilter` implements the selection logic. This choice makes `ParticleFilter` objects lightweight and provides a means of implementing a `State` instance-specific cache of `ParticleGroup` objects. The latter ensures that we do not create multiples of the same `ParticleGroup` which can occupy large amounts of memory. The caching also allows the creation of many of the same `ParticleFilter` object without needing to worry about memory constraints.

`ParticleFilter` can be subclassed (like `Trigger` and `Variant`), but only through the `CustomParticleFilter` class. This is necessary to prevent some internal details from leaking to the user. An example of a `CustomParticleFilter` that selects only particles with positive charge is given below.

```
from hoomd.filter import CustomParticleFilter

class PositiveCharge(CustomParticleFilter):
    def __init__(self, state):
        super().__init__(state)

    def __hash__(self):
        return hash(self.__class__.__name__)

    def __eq__(self, other):
        return type(self) == type(other)

    def find_tags(self, state):
        with state.cpu_local_snapshot as data:
            mask = data.particles.charge > 0
            return data.particles.tag[mask]
```

Custom Actions

In HOOMD-blue, we distinguish between the objects that perform an action on the simulation state (called *Actions*) and their containing objects that deal with setting state and the user interface (called *Operations*). Through composition, HOOMD-blue offers the ability to create custom actions in Python and wrap them in our `_CustomOperation` subclasses (divided on the type of action performed) allowing the execution of the action in the `Simulation` run loop. The feature makes user created actions behave indistinguishably from native C++ actions. Through custom actions, users can modify state, tune hyperparameters for performance, or observe parts of the simulation. In addition, we are adding a signal for Actions to send that would stop a `Simulation.run` call. This would allow actions to stop the simulation when they complete, which could be useful for tasks like tuning MC trial move sizes. With respect to performance,

with zero copy access to the data on the CPU or GPU, custom actions can also achieve high performance using standard Python libraries like NumPy, SciPy, Numba, CuPy and others. Below we show an example of an `Action` that switches particles of type `initial_type` to type `final_type` with a specified `rate` each time it is run. This action could be refined to implement a reactive MC move reminiscent of [GSJ] or to have a variable switch rate. These exercises are left to the reader.

```
import hoomd
from hoomd.filter import (
    Intersection, All, Type)
from hoomd.custom import Action

class SwapType(Action):
    def __init__(self, initial_type,
                 final_type, rate, filter=All()):
        self.final_type = final_type
        self.rate = rate
        self.filter = Intersection(
            [Type(initial_type), filter])

    def act(self, timestep):
        state = self._state
        final_type_id = state.particle_types.index(
            self.final_type)
        tags = self.filter(state)
        with state.cpu_local_snapshot as snap:
            tags = np.intersect1d(
                tags, snap.particles.tag, True)
            part = data.particles
            filtered_index = part.rtags[tags]
            N_swaps = int(len(tags) * self.rate)
            mask = np.random.choice(filtered_index,
                                    N_swaps,
                                    replace=False)
            part.typeid[mask] = final_type_id
```

Conclusion

With modern simulation analysis packages such as freud [RDH+], MDTraj [MBH+], and MDAnalysis [GLB+], [MDWB], initialization tools such as mbuild and foyer, and visualization packages like OVITO and plato [SD] using Python APIs, HOOMD-blue, built from the ground up with Python in mind, fits in seamlessly. Version 3.0 improves upon this and presents a Pythonic API that encourages customization. Through enabling Python subclassing of C++ classes, introducing custom actions, and exposing data in zero-copy arrays/buffers, we allow HOOMD-blue users to utilize the full potential of Python and the scientific Python community.

Acknowledgements

This research was supported by the National Science Foundation, Division of Materials Research Award # DMR 1808342 (HOOMD-blue algorithm and performance development) and by the National Science Foundation, Office of Advanced Cyberinfrastructure Award # OAC 1835612 (Pythonic architecture for MoSDeF). Hardware provided by NVIDIA Corp. is gratefully acknowledged. This research was supported in part through computational resources and services supported by Advanced Research Computing at the University of Michigan, Ann Arbor.

Appendix

In the appendix, we will provide more substantial applications of features new to HOOMD-blue.

Trigger that detects nucleation

This example demonstrates a `Trigger` that returns true when a threshold Q_6 Steinhardt order parameter [SNR] (as calculated by freud) is reached. Such a `Trigger` could be used for BCC nucleation detection which could trigger a decrease in cooling rate, a more frequent output of simulation trajectories, or any other desired action. Also, in this example we showcase the use of the zero-copy rank-local data access. This example also requires the use of ghost particles, which are a subset of particles bordering a MPI rank's local box. Ghost particles are known by a rank, but the rank is not responsible for updating them. In this case, ghost particles are required for computing the Q_6 value for particles near the edges of the current rank's local simulation box.

```
import numpy as np
import freud
from mpi4py import MPI
from hoomd.trigger import Trigger

class Q6Trigger(Trigger):
    def __init__(self, simulation, threshold,
                 mpi_comm=None):
        super().__init__()
        self.threshold = threshold
        self.state = simulation.state
        nr = simulation.device.num_ranks
        if nr > 1 and mpi_comm is None:
            raise RuntimeError()
        elif nr > 1:
            self.comm = mpi_comm
        else:
            self.comm = None
        self.q6 = freud.order.Steinhardt(l=6)

    def __call__(self, timestep):
        with self.state.cpu_local_snapshot as data:
            part = data.particles
            box = data.box
            aabb_box = freud.locality.AABBQuery(
                box, part.positions_with_ghosts)
            nlist = aabb_box.query(
                part.position,
                {'num_neighbors': 12,
                 'exclude_ii': True})
            Q6 = np.nanmean(self.q6.compute(
                (box, part.positions_with_ghosts),
                nlist).particle_order)
            if self.comm:
                return self.comm.allreduce(
                    Q6 >= self.threshold,
                    op=MPI.LOR)
            else:
                return Q6 >= self.threshold
```

Pandas Logger Back-end

Here we highlight the ability to use the `Logger` class to create a Pandas back end for simulation data. It will store the scalar and string quantities in a single `pandas.DataFrame` object while each array-like object is stored in a separate `DataFrame` object. All `DataFrame` objects are stored in a single dictionary.

```
import pandas as pd
from hoomd.custom import Action
from hoomd.util import (
    dict_flatten, dict_filter, dict_map)

def is_flag(flags):
    def func(v):
        return v[1] in flags
    return func
```

```
def not_none(v):
    return v[0] is not None

def hnd_2D_arrays(v):
    if v[1] in ['scalar', 'string', 'state']:
        return v
    elif len(v[0].shape) == 2:
        return {
            str(i): col
            for i, col in enumerate(v[0].T)}


class DataFrameBackEnd(Action):
    def __init__(self, logger):
        self.logger = logger

    def act(self, timestep):
        log_dict = self.logger.log()
        is_scalar = is_flag(['scalar', 'string'])
        sc = dict_flatten(dict_map(dict_filter(
            log_dict,
            lambda x: not_none(x) and is_scalar(x)),
            lambda x: x[0]))
        rem = dict_flatten(dict_map(dict_filter(
            log_dict,
            lambda x: not_none(x) \
                and not is_scalar(x)),
            hnd_2D_arrays))

        if not hasattr(self, 'data'):
            self.data = {
                'scalar': pd.DataFrame(
                    columns=[
                        '.'.join(k) for k in sc]),
                'array': {
                    '.'.join(k): pd.DataFrame()
                    for k in rem}}

        sdf = pd.DataFrame(
            {'.'.join(k): v for k, v in sc.items()},
            index=[timestep])
        rdf = {'.'.join(k): pd.DataFrame(
                    v, columns=[timestep]).T
            for k,v in rem.items()}
        data = self.data
        data['scalar'] = data['scalar'].append(sdf)
        data['array'] = {
            k: v.append(rdf[k])
            for k, v in data['array'].items()}
```

REFERENCES

[AGG] Joshua A. Anderson, Jens Glaser, and Sharon C. Glotzer. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. 173:109363. URL: http://www.sciencedirect.com/science/article/pii/S0927025619306627, doi:10.1016/j.commatsci.2019.109363.

[ALT] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. 227(10):5342–5359. URL: http://www.sciencedirect.com/science/article/pii/S0021999108000818, doi:10.1016/j.jcp.2008.01.047.

[AMS+] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. 1-2:19–25. URL: http://www.sciencedirect.com/science/article/pii/S2352711015000059, doi:10.1016/j.softx.2015.06.001.

[AW] B. J. Alder and T. E. Wainwright. Studies in Molecular Dynamics. I. General Method. 31(2):459–466. URL: https://aip.scitation.org/doi/abs/10.1063/1.1730376, doi:10.1063/1.1730376.

[BCR+] Surendra Byna, Jerry Chou, Oliver Rubel, Prabhat, Homa Karimabadi, William S. Daughter, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. doi:10.1109/SC.2012.92.

[BvdSvD] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. 91(1):43–56. URL: http://www.sciencedirect.com/science/article/pii/001046559500042E, doi:10.1016/0010-4655(95)00042-E.

[CCD+] David A. Case, Thomas E. Cheatham, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M. Merz, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J. Woods. The Amber biomolecular simulation programs. 26(16):1668–1688. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20290, doi:10.1002/jcc.20290.

[CG] Peter T Cummings and Justin B Gilmer. Open-source molecular modeling software in chemical engineering. 23:99–105. URL: http://www.sciencedirect.com/science/article/pii/S2211339819300073, doi:10.1016/j.coche.2019.03.008.

[DEG] Pablo F. Damasceno, Michael Engel, and Sharon C. Glotzer. Predictive Self-Assembly of Polyhedra into Complex Structures. 337(6093):453–457. URL: https://science.sciencemag.org/content/337/6093/453, arXiv:22837525, doi:10.1126/science.1220869.

[DZK+] Gregory L. Dignon, Wenwei Zheng, Young C. Kim, Robert B. Best, and Jeetain Mittal. Sequence determinants of protein phase behavior from a coarse-grained model. 14(1):e1005941. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005941, doi:10.1371/journal.pcbi.1005941.

[EGK+] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *Graph Drawing Software*, pages 127–148. Springer-Verlag.

[ESC+] Peter Eastman, Jason Swails, John D. Chodera, Robert T. McGibbon, Yutong Zhao, Kyle A. Beauchamp, Lee-Ping Wang, Andrew C. Simmonett, Matthew P. Harrigan, Chaya D. Stern, Rafal P. Wiewiora, Bernard R. Brooks, and Vijay S. Pande. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. 13(7):e1005659. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005659, doi:10.1371/journal.pcbi.1005659.

[FIE] Yue Fan, Takuya Iwashita, and Takeshi Egami. How thermally activated deformation starts in metallic glass. 5(1):1–7. URL: https://www.nature.com/articles/ncomms6083, doi:10.1038/ncomms6083.

[GKNV] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-phong Vo. A Technique for Drawing Directed Graphs. 19(3):214–230.

[GLB+] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. pages 98–105. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[GNA+] 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. 192:97–107. URL: http://www.sciencedirect.com/science/article/pii/S0010465515000867, doi:10.1016/j.cpc.2015.02.028.

[GSJ] Sharon C. Glotzer, Dietrich Stauffer, and Naeem Jan. Monte Carlo simulations of phase separation in chemically reactive binary mixtures. 72(26):4109–4112. URL: https://link.aps.org/doi/10.1103/PhysRevLett.72.4109, doi:10.1103/PhysRevLett.72.4109.

[GTK+] Horacio V. Guzman, Nikita Tretyakov, Hideki Kobayashi, Aoife C. Fogarty, Karsten Kreis, Jakub Krajniak, Christoph Junghans, Kurt Kremer, and Torsten Stuehn. ESPResSo++ 2.0: Advanced methods for multiscale molecular simulation. 238:66–76. URL: http://www.sciencedirect.com/science/article/pii/S0010465518304399, doi:10.1016/j.cpc.2018.12.017.

[Hun] John D. Hunter. Matplotlib: A 2D Graphics Environment. 9(3):90–95. doi:10.1109/MCSE.2007.55.

[JRM] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. Pybind11

– Seamless operability between C++11 and Python. URL: https://github.com/pybind/pybind11.

[KSJ+] Christoph Klein, János Sallai, Trevor J. Jones, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. In Randall Q Snurr, Claire S. Adjiman, and David A. Kofke, editors, *Foundations of Molecular Modeling and Simulation: Select Papers from FOMMS 2015*, Molecular Modeling and Simulation, pages 79–92. Springer. URL: https://doi.org/10.1007/978-981-10-1128-3_5, doi:10.1007/978-981-10-1128-3_5.

[KST+] Christoph Klein, Andrew Z. Summers, Matthew W. Thompson, Justin B. Gilmer, Clare McCabe, Peter T. Cummings, Janos Sallai, and Christopher R. Iacovella. Formalizing atom-typing and the dissemination of force fields with foyer. 167:215–227. URL: http://www.sciencedirect.com/science/article/pii/S0927025619303040, doi:10.1016/j.commatsci.2019.05.026.

[LPS] 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. Association for Computing Machinery. URL: https://doi.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[Mar] Marcus G. Martin. MCCCS Towhee: A tool for Monte Carlo molecular simulation. 39(14-15):1212–1222. URL: https://doi.org/10.1080/08927022.2013.828208, doi:10.1080/08927022.2013.828208.

[MBH+] 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. 109(8):1528–1532. URL: http://www.sciencedirect.com/science/article/pii/S0006349515008267, doi:10.1016/j.bpj.2015.08.015.

[McK] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61. URL: https://conference.scipy.org/proceedings/scipy2010/mckinney.html, doi:10.25080/Majora-92bf1922-00a.

[MDWB] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. 32(10):2319–2327. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[MRR+] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. 21(6):1087–1092. URL: https://aip.scitation.org/doi/abs/10.1063/1.1699114, doi:10.1063/1.1699114.

[NBB+] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W. Glass, Hans Hasse, Jadran Vrabec, and Martin Horsch. Ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems. 10(10):4455–4464. URL: https://doi.org/10.1021/ct500169q, doi:10.1021/ct500169q.

[PGM+] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\ textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc. URL: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Pli] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. URL: https://www.osti.gov/biblio/10176421, doi:10.2172/10176421.

[PVG+] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. 12(85):2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[RDH+] 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. page 107275. URL: http://www.sciencedirect.com/science/article/pii/S0010465520300916, doi:10.1016/j.cpc.2020.107275.

[SCW] Romelia Salomon-Ferrer, David A. Case, and Ross C. Walker. An overview of the Amber biomolecular simulation package. 3(2):198–210. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1121, doi:10.1002/wcms.1121.

[SD] Matthew Spellings and Bradley D. Dice. Plato. URL: https://github.com/glotzerlab/plato.

[SDS+] David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1–11. Association for Computing Machinery. URL: https://doi.org/10.1145/1654059.1654126, doi:10.1145/1654059.1654126.

[SMAG] Matthew Spellings, Ryan L. Marson, Joshua A. Anderson, and Sharon C. Glotzer. GPU accelerated Discrete Element Method (DEM) molecular dynamics for conservative, faceted particle simulations. 334:460–467. URL: http://www.sciencedirect.com/science/article/pii/S0021999117300244, doi:10.1016/j.jcp.2017.01.014.

[SMM+] Jindal K. Shah, Eliseo Marin-Rimoldi, Ryan Gotchy Mullen, Brian P. Keene, Sandip Khan, Andrew S. Paluch, Neeraj Rai, Lucienne L. Romanielo, Thomas W. Rosch, Brian Yoo, and Edward J. Maginn. Cassandra: An open source Monte Carlo package for molecular simulation. 38(19):1727–1739. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.24807, doi:10.1002/jcc.24807.

[SNR] Paul J. Steinhardt, David R. Nelson, and Marco Ronchetti. Bond-orientational order in liquids and glasses. 28(2):784–805. URL: https://link.aps.org/doi/10.1103/PhysRevB.28.784, doi:10.1103/PhysRevB.28.784.

[Sto] John Edward Stone. An effiicient library for parallel ray tracing and animation. URL: http://jedi.ks.uiuc.edu/~johns/tachyon/papers/thesis.pdf.

[Stu] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. 18(1):015012. URL: https://doi.org/10.1088%2F0965-0393%2F18%2F1%2F015012, doi:10.1088/0965-0393/18/1/015012.

[TGM+] Matthew W. Thompson, Justin B. Gilmer, Ray A. Matsumoto, Co D. Quach, Parashara Shamaprasad, Alexander H. Yang, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. Towards molecular simulations that are transparent, reproducible, usable by others, and extensible (TRUE). 118(9-10):e1742938. URL: https://doi.org/10.1080/00268976.2020.1742938, doi:10.1080/00268976.2020.1742938.

[vdWCV] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. 13(2):22–30. doi:10.1109/MCSE.2011.37.

[VGO+] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. SciPy 1.0: Fundamental algorithms for scientific computing in Python. 17(3):261–272. URL: https://www.nature.com/articles/s41592-019-0686-2, doi:10.1038/s41592-019-0686-2.

[WWS+] Florian Weik, Rudolf Weeber, Kai Szuttor, Konrad Breitsprecher, Joost de Graaf, Michael Kuron, Jonas Landsgesell, Henri Menke, David Sean, and Christian Holm. ESPResSo 4.0 – an extensible software package for simulating soft matter systems. 227(14):1789–1816. URL: https://doi.org/10.1140/epjst/e2019-800186-9, doi:10.1140/epjst/e2019-800186-9.

[zot] CuPy. URL: https://cupy.chainer.org/.

Compyle: a Python package for parallel computing

Aditya Bhosale[‡§], Prabhu Ramachandran[‡§*]

Abstract—Compyle allows users to execute a restricted subset of Python on a variety of HPC platforms. It is an embedded domain-specific language (eDSL) for parallel computing. It currently supports multi-core execution using Cython, and OpenCL and CUDA for GPU devices. Users write code in a restricted subset of Python that is automatically transpiled to high-performance Cython or C. Compyle also provides a few very general purpose and useful parallel algorithms that allow users to write code once and have them run on a variety of HPC platforms.

In this article, we show how to implement a simple two-dimensional molecular dynamics (MD) simulation package in pure Python using Compyle. The result is a fully parallel program that is relatively easy to implement and solves a non-trivial problem. The code transparently executes on multi-core CPUs and GPGPUs allowing simulations with millions of particles. A 3D MD code is also provided and compares very favorably with a well known, open source, molecular dynamics package.

Index Terms—High-performance computing, multi-core CPUs, GPGPU accelerators, parallel algorithms, transpilation

Motivation and background

In this brief article we provide an overview of Compyle (https://compyle.rtfd.io). Compyle is a BSD licensed, Python package that allows users to write code once in pure Python and have it execute transparently on both multi-core CPUs or GPGPUs via CUDA or OpenCL. Compyle is available on PyPI and hosted on github at https://github.com/pypr/compyle

Users often write their code in one language (sometimes a high-performance language), only to find out later that the same performance is not possible on newer hardware without making significant changes. For example, many scientists do not make use of GPGPU hardware despite their excellent performance and availability. One of the problems is that it is often hard to reuse code developed in one language and expect it to work on all of the platforms. Moreover, GPUs are parallel machines and extracting performance from them requires the use of parallel algorithms. Unless the initial development is done with this in mind, one cannot easily convert a serial code into a parallel one.

There are many powerful tools available in the Python ecosystem today that facilitate high-performance computing. PyPy is a Python implementation in Python that features a JIT that allows one to execute pure Python code at close to C-speeds. Numba uses the LLVM compiler infrastructure to generate machine code that can rival native C code. Numba also supports execution on GPUs. There are also compilers like Pythran that transpile a subset of Python to C++ and support multi-core execution using OpenMP. Cython is a much used and mature compiler that makes it possible to write code in a mixture of Python and C. Cython also provides loop parallelism using OpenMP. Packages like cppimport and pybind11 make it a breeze to integrate Python with C++ code. In addition, there are powerful interfaces to GPUs via packages like PyOpenCL or PyCUDA. Furthermore, packages like Reikna provide an abstraction and higher level API using PyOpenCL and PyCUDA. Of these, Numba has matured a great deal and is both easy to use and versatile.

Given this context, one may wonder why Compyle exists at all. While Compyle grew out of a project that pre-dates Numba, the real reason that Compyle exists is that solves a different problem from most of the existing tools. Understanding this requires a bit of a context. As a prototypical example, we look at a simple molecular dynamics simulation where N particles interact with each other via a Lennard-Jones potential. This problem is discussed at length in [Sch15].

In order to implement this, the typical workflow for a Python programmer would be to prototype the molecular dynamics simulation code in pure Python and obtain a proof of concept. One would then optimize this code so as to run larger problems in a smaller amount of time. Very often this would mean changing some data structures, writing vectorized code using NumPy arrays, and then resorting to tools like Numba to extract even more performance (sometimes this requires that the code be devectorized to make the looping explicit). Numba is an impressive tool and one could say almost works magically well. In fact, for some problems it will even do a good job of parallelizing the code to run on multiple cores. However, one cannot execute this same code on a GPU without making significant modifications, to the point of practically rewriting it. While Numba offers some help here with the CUDA and ROCm support, one would still have to change quite a lot of code to have it work on these architectures. As such, the issue is that it is difficult to have the same Python code execute well on CPUs and GPUs.

The reason for this difficulty is that GPUs are inherently parallel with many thousands of cores. Writing code to effectively use such hardware requires a significant re-think of the algorithms used. In particular the algorithm has to be fully parallelized. While this is easy to do for simple problems, most useful computational codes involve non-trivial algorithms, which are not always easy to parallelize.

What Compyle attempts to do is to allow one to write code **once** in a highly restrictive subset of pure Python and have this run in parallel on both CPUs and GPUs. This is a significant

‡ *Department of Aerospace Engineering*
§ *IIT Bombay, Mumbai, India*
* *Corresponding author: prabhu@aero.iitb.ac.in*

difference from all the tools that we have mentioned above.

The difficulty in doing this is that it does require a change in approach and also a loss of the typical conveniences with high-level Python. While Compyle does not allow arbitrary Python code, since the code is still written in Python and not another language, it makes it much easier for users to write and manage the code.

Compyle provides important parallel programming algorithms that one typically requires when writing parallel programs. These are the element-wise operations (or maps), reductions, and parallel prefix scans. These primitives are written such that the same program can be executed on both multi-core CPUs and GPUs with minimal or no changes to the code.

This is currently not possible with any of the other tools. In addition, Compyle has the following features:

- Generates either Cython or ANSI C code depending on the backend and this code is quite readable (to a user familiar with Cython or C). This makes it much easier to understand and debug.
- Designed to be relatively easy to use as a code generator.
- Support for templated code generation to minimize repetitive code.
- Highly restrictive language that facilitates cross-platform execution.

Compyle is in principle very similar to the copperhead package described in [CGK11]. The design of copperhead is very elegant. However, it appears that copperhead is no longer under development, the package has no commits after 2013 and is not available on PyPI (another unrelated package with the same name is available). While it does support execution via C++ and CUDA, it does not support OpenCL. We were not aware of copperhead until very recently and are likely to try and incorporate ideas from it into Compyle.

Compyle is actively used by a non-trivial, open source, SPH framework called PySPH and discussed in some detail in [RP+19] and [Ram16]. Compyle makes it possible for users to write their SPH codes in high-level Python and have it executed on multi-core and GPU accelerators with negligible changes to their code. Unfortunately, Compyle is not used much outside of this context, so while it does solve many problems, it is still under heavy development.

In this paper we write a simple two-dimensional molecular dynamics system that is described and discussed in the article by [Sch15]. Our goal is to implement this system in pure Python using Compyle. Through this we demonstrate the ease of use and power of Compyle. We write programs that execute efficiently in parallel on CPUs and GPUs without any modifications. We use this as a way to illustrate the three important parallel algorithms and show how they allow us to solve non-trivial problems. A three-dimensional version is also implemented and compared with HooMD. The results show that our code can be almost two-times faster for the problem considered. A Google Colaboratory notebook is provided to make it easy to explore Compyle and these examples.

High-level overview

We now provide a high-level overview of Compyle and its basic approach. This is helpful when using Compyle.

It is important to keep in mind that Compyle does **not** provide a greater abstraction of the hardware but allows a user to write code in pure Python and have that same code execute on multiple different platforms. We currently support multi-core execution using OpenMP and Cython, and also transparently support OpenCL and CUDA so the same could could potentially be reused on a GPGPU. Compyle makes this possible by providing three important parallel algorithms, an elementwise operation (a parallel map), a parallel scan (also known as a prefix sum), and a parallel reduction. The Cython backend provides a native implementation whereas the OpenCL and CUDA backend simply wrap up the implementation provided by PyOpenCL and PyCUDA. These three algorithms make it possible to write a variety of non-trivial parallel algorithms for high performance computing. Compyle also provides the ability to write custom kernels with support for local/shared memory specifically for OpenCL and CUDA backends. Compyle provides simple facilities to annotate arguments and types and can optionally make use of Python 3's type annotation feature as well. Compyle also features JIT compilation and automatic type inference.

Compyle does not provide support for any high level Python and only works with a highly restricted Python syntax. While this is not very user-friendly, we find that in practice this is vitally important as it ensures that the code users write will run efficiently and seamlessly execute on both a CPU and a GPU with minimum or ideally no modifications. In addition, Compyle allows users to generate code using mako templates in order to maximize code reuse. Since Compyle performs source transpilation, it is also possible to use Compyle as a code-generation engine and put together code from pure Python to build fairly sophisticated computational engines.

The functionality that Compyle provides falls broadly in two categories,

- Common parallel algorithms that will work across backends. This includes, elementwise operations, reductions, and prefix-sums/scans.
- Specific support to run code on a particular backend. This is for code that will only work on one backend by definition. This is necessary in order to best use different hardware and also use differences in the particular backend implementations. For example, the notion of local (or shared) memory only has meaning on a GPGPU. In this category we provide support to compile and execute Cython code, and also create and execute a GPU kernel. These features are not discussed in this article.

In general the subset of Python that Compyle supports are:

- Functions with a C-syntax, this means no default or keyword arguments.
- Function arguments may be declared using either type annotation or using a decorator or with default arguments (which are only used to suggest the type).
- No Python data structures, i.e. no lists, tuples, sets, or dictionaries.
- Contiguous Numpy arrays are supported but must be one dimensional and must be a numerical data type.
- No memory allocation is allowed inside these functions.
- On OpenCL no recursion is supported but this will work with Cython or CUDA.
- Currently, all function calls must not use dotted names, i.e. don't use `math.sin`, instead just use `sin`. This is because we do not perform any kind of name mangling of the generated code to make it easier to read.

- Compyle does support JIT compilation. If the type annotation is not explicitly supplied, the types can be automatically inferred when the functions are called.
- No support for classes and structs although this may change in a future release.

In what follows we provide a high-level introduction to the basic parallel algorithms in the context of the prototypical molecular dynamics problem. By the end of the article we show how easy it is to write the code with Compyle and have it execute on multi-core CPUs and GPGPUs. The programs we document here are also available as part of the Compyle examples. We provide a convenient Google Colaboratory notebook where users can run the simple examples on a GPU as well.

Installation

Installation of Compyle is by itself straightforward and this can be done with pip using:

```
pip install compyle
```

For execution on a CPU, Compyle depends on Cython and a C++ compiler on the local machine. Multi-core execution requires OpenMP to be available. Detailed instructions for installation are available at the compyle installation documentation. For execution on a GPU Compyle requires that either PyOpenCL or PyCUDA be installed. It is possible to install the required dependencies using the extras argument as follows:

```
pip install compyle[opencl]
```

Compyle is still under heavy development and one can install the package using a git checkout from the repository on github at https://github.com/pypr/compyle

Parallel algorithms

We will work through a molecular dynamics simulation of N particles using the Lennard-Jones potential energy for interaction. Each particle interacts with every other particle and together the system of particles evolves in time. The Lennard-Jones potential energy is given by,

$$u(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right)$$

Each particle introduces an energy potential and if another particle is at a distance of r from it, then the potential experienced by the particle is given by the above equation. The gradient of this potential energy function produces the force on the particle. Therefore if we are given two particles at positions, $\vec{r}_i$ and $\vec{r}_j$ respectively then the force on the particle j is dependent on the value of $|\vec{r}_j - \vec{r}_i|$ and the gradient is:

$$\vec{F}_{i \leftarrow j} = \frac{24\varepsilon}{r_{ij}^2}\left(2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right)\vec{r}_{ij}$$

Where $r_{ij} = |\vec{r}_{ij}|$ and $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$. The left hand side is the force on particle i due to particle at j. Here, we use $\sigma = \varepsilon = m = 1$ for our implementation. We use the velocity Verlet algorithm in order to integrate the system in time. We use a timestep of Δt and as outlined in [Sch15], the position and velocity of the particles are updated in the following sequence:

1) Positions of all particles are updated using the current velocities as $x_i = x_i + v_i\Delta t + \frac{1}{2}a_i\Delta t^2$. The velocities are then updated by half a step as $v_i = v_i + \frac{1}{2}a_i\Delta t$.
2) The new acceleration of all particles are calculated using the updated positions.
3) The velocities are then updated by another half a step.

In the simplest implementation of this, all particles influence all other particles. This can be implemented very easily in Python and Compyle. Our implementation will be parallel from the get-go and will work on both CPUs and GPUs.

Once we complete the simple implementation we consider a very important performance improvement where particles that are beyond 3 natural units, i.e. $r_{ij} > 3$ do not influence each other (beyond this distance the force is negligible). This can be used to reduce the complexity of the computation of the mutual forces from an $O(N^2)$ to an $O(N)$ computation. However, implementing this easily in parallel is not so straightforward.

Due to the simplicity of the initial implementation, all of these steps can be implemented using what are called "elementwise" operations. This is the simplest building block for parallel computing and is also known as the "parallel map" operation.

Elementwise

An elementwise operation can be thought of as a parallel for loop. It can be used to map every element of an input array to a corresponding output. Here is a simple elementwise function implemented using Compyle to execute step 1 of the above algorithm.

```
@annotate(float='m, dt',
          gfloatp='x, y, vx, vy, fx, fy')
def integrate_step1(i, m, dt, x, y, vx, vy, fx, fy):
    axi, ayi = declare('float', 2)
    axi = fx[i] / m
    ayi = fy[i] / m
    x[i] += vx[i] * dt + 0.5 * axi * dt * dt
    y[i] += vy[i] * dt + 0.5 * ayi * dt * dt
    vx[i] += 0.5 * axi * dt
    vy[i] += 0.5 * ayi * dt
```

The annotate decorator is used to specify types of arguments and the declare function is used to specify types of variables declared in the function. In this case, `gfloatp` indicates a global double pointer data type. Compyle also supports Python3 style type annotations using the types defined in `compyle.types`.

Specifying types can be avoided by using the JIT compilation feature which infers the types of arguments and variables based on the types of arguments passed to the function at runtime. Following is the implementation of steps 2 and 3 without the type declarations.

```
@annotate
def calculate_force(i, x, y, fx, fy, pe,
                    num_particles):
    force_cutoff = 3.
    force_cutoff2 = force_cutoff * force_cutoff
    for j in range(num_particles):
        if i == j:
            continue
        xij = x[i] - x[j]
        yij = y[i] - y[j]
        rij2 = xij * xij + yij * yij
        if rij2 > force_cutoff2:
            continue
        irij2 = 1.0 / rij2
        irij6 = irij2 * irij2 * irij2
        irij12 = irij6 * irij6
        pe[i] += (2 * (irij12 - irij6))
        f_base = 24 * irij2 * (2 * irij12 - irij6)

        fx[i] += f_base * xij
```

```
        fy[i] += f_base * yij

@annotate
def integrate_step2(i, m, dt, x, y, vx, vy, fx, fy):
    vx[i] += 0.5 * fx[i] * dt / m
    vy[i] += 0.5 * fy[i] * dt / m
```

Finally, these components can be brought together to write the step functions for our simulation,

```
@annotate
def step_method1(i, x, y, vx, vy, fx, fy, pe, xmin,
                 xmax, ymin, ymax, m, dt,
                 num_particles):
    integrate_step1(i, m, dt, x, y, vx, vy, fx, fy)


@annotate
def step_method2(i, x, y, vx, vy, fx, fy, pe, xmin,
                 xmax, ymin, ymax, m, dt,
                 num_particles):
    calculate_force(i, x, y, fx, fy, pe,
                    num_particles)
    integrate_step2(i, m, dt, x, y, vx, vy, fx, fy)
```

These can then be wrapped using the `Elementwise` class and called as normal python functions.

```
step1 = Elementwise(step_method1,
                    backend=self.backend)
step2 = Elementwise(step_method2,
                    backend=self.backend)
```

One can also use the `@elementwise` decorator on the step functions and those can then be directly called without having to wrap them using `Elementwise`.

Note that in the above, `step_method1, step_method2` are the ones that are wrapped into an elementwise operation. The `integrate_step` methods are merely called by these. For an elementwise kernel, the first argument is always the index of the particular element being processed, in this case `i`. One can think of the function as the block of code being executed by a `for` loop. The number of elements iterated over is always implicitly based on the first array argument passed to the function, in this case, `x`.

The simulation can then be executed simply as follows,

```
# Initialize x, y
# Initialize vx, vy, fx, fy, pe to zeros

num_steps = int(t // dt)
for i in range(num_steps):
    step1(x, y, vx, vy, fx, fy, pe, xmin, xmax,
          ymin, ymax, m, dt, self.num_particles)
    step2(x, y, vx, vy, fx, fy, pe, xmin, xmax,
          ymin, ymax, m, dt, self.num_particles)
```

We have used a fixed wall non-periodic boundary condition for our implementation. The details on the implementation of the boundary condition can be found in the example section of Compyle's github repository here.

The backend used can be changed using the following code:

```
from compyle.api import get_config
# On OpenMP
get_config().use_openmp = True

# Run with OpenCL
get_config().use_opencl = True
```

No other code changes are needed.

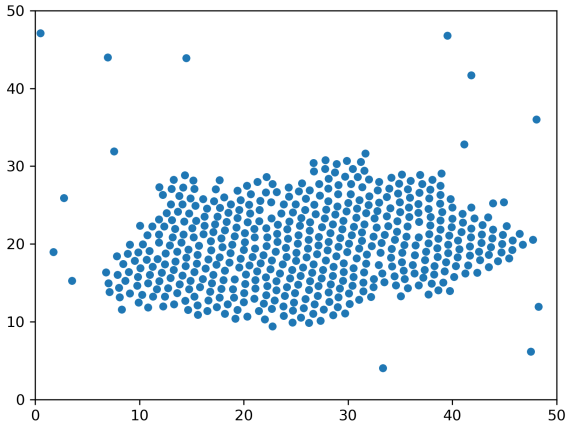


Fig. 1: *Snapshot of simulation with 500 particles.*

Reduction

To check the accuracy of the simulation, the total energy of the system can be monitored. The total energy for each particle can be calculated as the sum of its potential and kinetic energy. The total energy of the system can then be calculated by summing the total energy over all particles.

The reduction operator reduces an array to a single value. Given an input array $(a_0, a_1, a_2, \cdots, a_{n-1})$ and an associative binary operator $\oplus$, the reduction operation returns the value $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$.

Compyle also allows users to give a map expression to map the input before applying the reduction operator. The total energy of our system can thus be found as follows using reduction operator in Compyle.

```
@annotate
def calculate_energy(i, vx, vy, pe, num_particles):
    ke = 0.5 * (vx[i] * vx[i] + vy[i] * vy[i])
    return pe[i] + ke

energy_calc = Reduction('a+b',
                        map_func=calculate_energy,
                        backend=backend)
total_energy = energy_calc(vx, vy, pe, num_particles)
```

Here, in the expression `'a+b'` `a` represents a_i and `b` represents the reduction result till $i-1$, i.e. $\sum_0^{i-1} a_k$. For the maximum for example one would write `'max(a, b)'`. Common reductions like sum, max and min are also available but we show the general form above where we can also map the values using the function given before the reduction is applied.

Initial Results

Figure 1 shows a snapshot of simulation using 500 particles and bounding box size 50 with a non-periodic boundary condition.

For evaluating our performance, we ran our implementation on a 2.9 Ghz quad-core Intel Core i7 processor and an NVIDIA Tesla P100 GPU. We used $dt = 0.02$ and ran the simulation for 25 timesteps. Figures 2 and 3 show the speedup achieved over serial execution using Cython by using OpenMP, OpenCL and CUDA. As you can see on the CPUs we get more than a 5x speedup (despite having only 4 cores). However, on the GPU we get around a 200x speedup. This is compared to very fast execution on a single

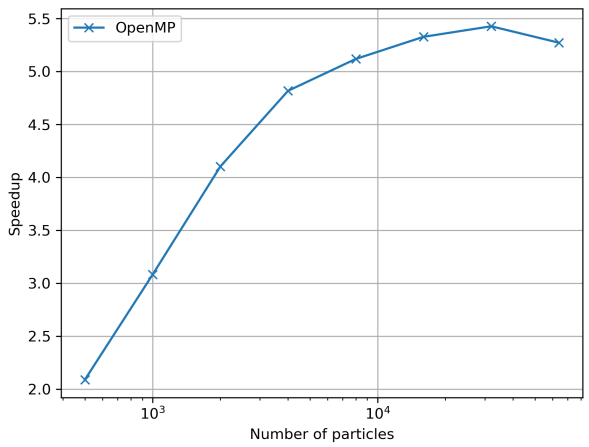


Fig. 2: *Speed up over serial Cython using OpenMP.*

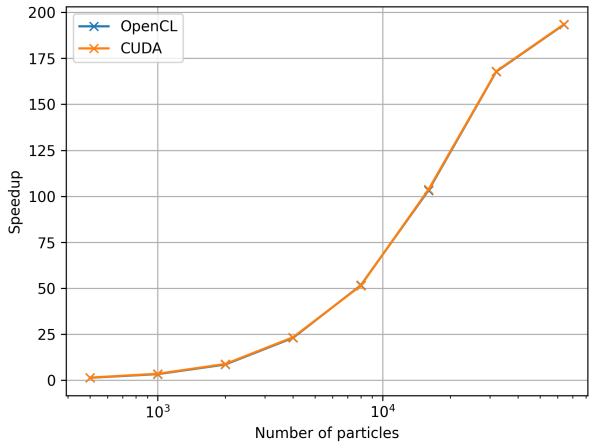


Fig. 3: *Speed up over serial Cython using CUDA and OpenCL.*

Intel Xeon 2.3GHz CPU. The fact that we can use both OpenCL and CUDA is also very important as on some operating systems, there is no CUDA support even though OpenCL is supported (like the GPUs on MacOS). Note that by default Compyle uses floating point precision on the GPUs as most GPUs perform much better with floating point precision. We can use double precision on the GPU using `get_config().use_double = True` if we require it. Again, we do not need to change the solver to do this. Our implementation is about 2x slower when using double precision on an NVIDIA Tesla P100 GPU which is typically expected.

This is in itself remarkable given that all we do to run on the GPU or CPU is to simply set the appropriate backend. In most of the Compyle examples, we use a command line argument to switch the backend. So with exactly the same code we are able to immediately run our program fully in parallel and have it run on both multi-core CPUs as well as GPUs.

Many problems can be solved using the map-reduce approach above. However, almost all non-trivial applications require a bit more than that and this is where the parallel scan becomes very important.

Scans

Up to now we have found the influence of all particles on each other. Since the force on two particles is negligible when they are more than 3 units apart, we do not have to loop over all the particles, we can therefore reduce the computation to an $O(N)$ computation and increase performance significantly. One way of doing this is to bin the particles into small boxes and given a particle in a box, only interact with the box and its nearest neighbor boxes.

Implementing this in serial is fairly easy, but if we want this to work fast and scale on a GPU we must implement a parallel algorithm. This is where the parallel scan comes in and why this parallel algorithm is so important. The parallel prefix scan is described in detail in the excellent article by Blelloch [Ble90]. Compyle provides an implementation of the scan algorithm on the CPU and the GPU.

Since the scan algorithm is a bit more complex and most folks are unfamiliar with it, we first provide a simpler example application that we solve and then move back to our molecular dynamics application.

Scans are generalizations of prefix sums / cumulative sums and can be used as building blocks to construct a number of parallel algorithms. These include but not are limited to sorting, polynomial evaluation, and tree operations.

Given an input array $a = (a_0, a_1, a_2, \cdots, a_{n-1})$ and an associative binary operator $\oplus$, a prefix sum operation returns the following array

$$y = (a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}))$$

The scan semantics in Compyle are similar to those of the `GenericScanKernel` in PyOpenCL. This allows us to construct generic scans by having an input expression, an output expression and a scan operator. The input function takes the input array and the array index as arguments and can be used to map the input array before running the scan. The output expression can then be used to map and write the scan result as required. The output function also operates on the input array and an index but also has the scan result, the previous item and the last item in the scan result available as arguments.

Below is an example of implementing a parallel "where". This returns elements of an array where a given condition is satisfied. The following example returns elements of the array that are smaller than 50.

```
@annotate
def input_expr(i, ary):
    return 1 if ary[i] < 50 else 0

@annotate
def output_expr(i, prev_item, item, N, ary, result,
                result_count):
    if item != prev_item:
        result[item - 1] = ary[i]
    if i == N - 1:
        result_count[0] = item

ary = np.random.randint(0, 100, 1000, dtype=np.int32)
res = np.zeros(len(ary.data), dtype=np.int32)
count = np.zeros(1, dtype=np.int32)

res, count, ary = wrap(res, count, ary, backend=backend)
scan = Scan(input_expr, output_expr, 'a+b',
            dtype=np.int32, backend=backend)
scan(ary=ary, result=res, result_count=count)
res.pull()
```

```
count.pull()
count = count.data[0]
res = res.data[:count]
```

The argument `i`, similar to that seen in elementwise kernels is the current index, the argument `item` is the result of the scan including the input at index `i`. The `prev_item` is the result of the array at index `i-1`. `item` and `prev_item` are reserved variables and users should not use them when writing the input and output functions.

In the above example, the input expression returns 1 only when the value at index `i` is less than 50. So as long as the array elements are greater than 50, the value of `item` will remain the same and will only increase when an element less than 50 is found at the index. Thus, the condition `item != prev_item` will only be satisifed for indices at which the value of `ary[i]` is less than 50.

The `input_expr` could also be used as the map function for reduction and the required size of result could be found before running the scan and the result array can be allocated accordingly.

Back to the MD problem

To reduce the complexity of the problem from $O(N^2)$ to $O(N)$, we use a binning strategy as mentioned in the previous section. We partition our domain into square bins of size 3 units. Then for each particle, all the particles within a radius of 3 units from it will lie inside of the 9 neighboring bins. For a bin with coordinates $c = (m,n)$, these 9 bins will be,

$$N(c) = \{c + d \mid d \in \{-1,0,1\} \times \{-1,0,1\}\}$$

The idea is that for each particle we will iterate over all particles in these 9 bins and check if the distance between the particle and the query particle is less than 3. The inter-particle force will be computed only then between the two particles. This algorithm is often called a nearest-neighbor particle search (NNPS) algorithm. To implement this, we first find the bin to which each particle belongs. This is done as follows,

$$c = \left(\left\lfloor \frac{x}{h} \right\rfloor, \left\lfloor \frac{y}{h} \right\rfloor\right)$$

where x and y are the coordinates of the particle and h is the required radius which in our case is 3. Note that our problem is setup such that the left bottom corner is at the origin. We then flatten these bin coordinates to map each bin to a unique integer we call the 'key'. We sort these keys and an array of indices of the particles such that the sorted indices have all particles in the same cell as contiguous elements. Compyle provides a sort function which uses the PyOpenCL radix sort for OpenCL backend, thrust sort for the CUDA backend and simple numpy sort for the cython backend.

To find the particles belonging to the 9 neighboring bins, we now need to find the index in the sorted indices array at which each key starts. This can be found in parallel using a scan as follows,

```
@annotate
def input_scan_keys(i, keys):
    return 1 if i == 0 or keys[i] != keys[i - 1] \
        else 0


@annotate
def output_scan_keys(i, item, prev_item, keys,
                     start_indices):
    key = keys[i]
    if item != prev_item:
        start_indices[key] = i
```

Once we have the start indices array, we can also find the number of particles in each bin using a simple elementwise operation as follows,

```
@annotate
def fill_bin_counts(i, keys, start_indices,
                    bin_counts, num_particles):
    if i == num_particles - 1:
        last_key = keys[num_particles - 1]
        bin_counts[last_key] = num_particles - \
                start_indices[last_key]
    if i == 0 or keys[i] == keys[i - 1]:
        return
    key = keys[i]
    prev_key = keys[i - 1]
    bin_counts[prev_key] = start_indices[key] - \
            start_indices[prev_key]
```

Now we can iterate over all neighboring 9 bins, find the key corresponding to each of them, then lookup the start index for that key in the `start_indices` array and the number of particles in the cell by looking up in the `bin_counts` array. Then lookup the sorted indices array to find the indices of the particles belonging to these bins and find the particles within a distance of 3 units.

However, note that we still have a challenge in storing these neighboring particles as we do not know the number of neighboring particles beforehand and so cannot allocate an array of that size. Moreover, since each particle can have different number of neighbors, it is also not straightforward to know where in the neighbors array we need to look to find the neighbors of a particular particle.

We use a two pass approach to solve this problem. In the first pass we find the number of neighbors for each particle. We then run a scan over this array to find the start indices for neighbors of each particle in the neighbors array as follows,

```
@annotate
def input_start_indices(i, counts):
    return 0 if i == 0 else counts[i - 1]


@annotate
def output_start_indices(i, item, indices):
    indices[i] = item
```

We then allocate the neighbors array of size equal to sum of all neighbor lengths. The second pass is then another elementwise operation where each particle writes its neighbors starting from the start index calculated from the scan.

More details on this implementation can be found in the examples section of our repository here. We have also implemented a more efficient version of the nearest neighbor searching algorithm using a counting sort instead of the radix sort which is 30% faster that can be found here.

Performance comparison

Figure 4 shows the speedup achieved by the OpenCL and CUDA backends running on a GPU relative to serial code running using Cython (on a single CPU core) for the linear version of the algorithm. Figure 5 shows the time taken for these simulations. It can be seen that the algorithm is linear for large values of number of particles. We again get more than a 100x speedup using the GPU over a single CPU core. Note that on the NVIDIA P100 GPU we are able to run a simulation with 25 timesteps for 5 million particles in less than a second, showing the excellent performance attained.

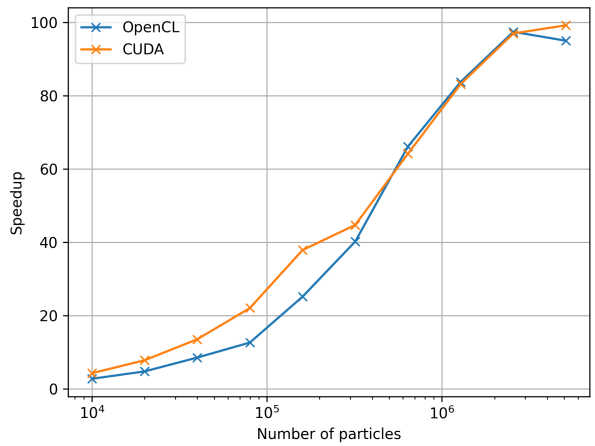


Fig. 4: *Speed up over serial cython using CUDA and OpenCL using the NNPS.*

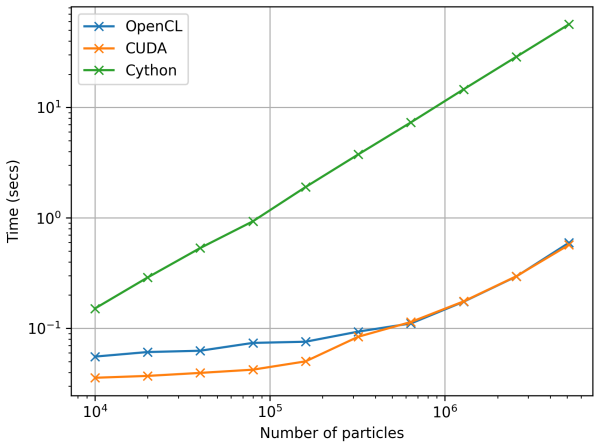


Fig. 5: *Time taken for simulation using serial cython, CUDA and OpenCL.*

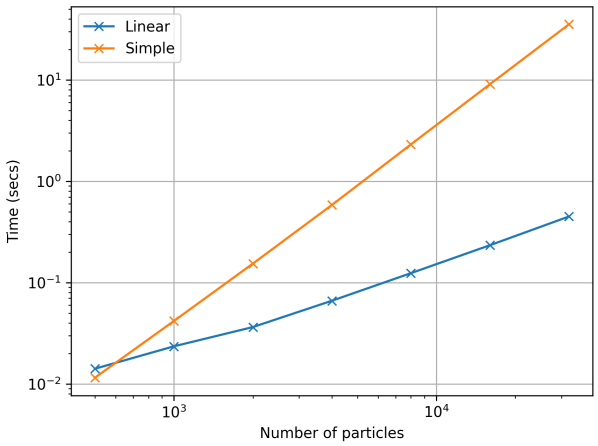


Fig. 6: *Time taken for simulation using $O(N)$ (Linear) and $O(N^2)$ (Simple) approach.*

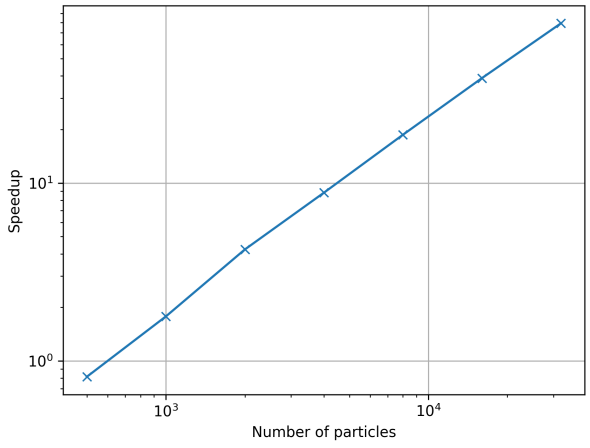


Fig. 7: *Speed up using $O(N)$ over $O(N^2)$ approach.*

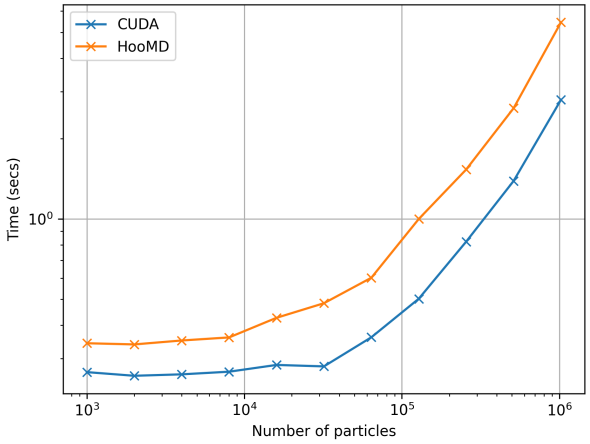


Fig. 8: *Time taken for HooMD and our implementation using CUDA backend.*

Figure 6 shows the time taken for simulation using $O(N)$ and $O(N^2)$ approach. Figure 7 shows the speed up acheived by using the $O(N)$ algorithm as compared to the $O(N^2)$ algorithm using the serial cython backend. We have about a 100 fold speed up with the improved algorithm for only 32,000 particles.

The performance of the algorithm can be further improved by aligning the x and y coordinate arrays according to the sorted indices. This will improve the global memory access pattern on the GPU giving a better performance. This can be done easily in Compyle using `compyle.array.align` which uses a single elementwise operation to align multiple arrays in a given order. We have not explored this in this paper.

We have also implemented a 3D version of the simulation with both periodic and non-periodic boundary conditions. We compared our implementation with HooMD for a 3D periodic simulation on an NVIDIA Tesla P100 GPU. Figure 8 shows the results of this comparison. We found our implementation to be about 2x faster than HooMD. To check the correctness of our implementation, we have also provided a script to generate plots of potential and kinetic energy of the system at every 100 timesteps using HooMD and our implementation.

All of the code discussed above is available in the examples directory of the Compyle repository here. All of the code, with two different NNPS implementations, and featuring a command line interface, comes to around 500 lines of code. This is quite exciting as this code can be executed on either a multi-core CPU or a GPU with no code changes.

Limitations

While Compyle is really powerful and convenient, it does use a rather verbose and low-level syntax. In practice we have found that this is not a major problem. The more serious issue is the fact that we cannot directly use external libraries in a platform neutral way. For example, there are ways to use an external OpenCL or CUDA library but this will not be usable on a CPU. Obviously one cannot use normal Python code and use basic Python data structures. This is because the Python data structures would need to be implemented in the target language. Furthermore, one cannot use well established libraries like scipy from within the parallel constructs. The reason for this is that scipy and other libraries are not necessarily available for use on a GPU or even on multi-core CPUs. These are limitations that are beyond the scope of Compyle at this point.

The low-level API that Compyle provides turns out to be quite an advantage as Compyle code is usually very fast the first time it runs. This is because it will refuse to run any code that uses Python objects. By forcing the user to write the algorithms conforming to the constraints makes the code efficient. It also forces the user to think along the lines of parallel algorithms. This is a major factor. We have used Compyle in the context of a larger scientific computing project and have found that while the limitations are annoying, the benefits are generally worth it.

Compyle has also only been used in the context of the PySPH project and as such has not seen a lot of community adoption. This has meant that there are many rough edges. We are hoping to improve the package and are also hopeful for community contributions eventually.

Future work

There are several improvements that are planned for Compyle.

- Some internal cleanup is necessary. This is especially true of the Cython backend which has grown organically and requires a reimplementation.
- Many of the CPU related algorithms, like sorting, and many of the reductions are still serial. These are relatively easy to fix.
- The Cython backend may be eventually replaced using pybind11 if possible.
- The API requires some cleanup in many places. We also hope to look at the copperhead package to improve our API.
- While Compyle does support simple structs, this API is still not clean enough to be used in general.
- We also hope to add support for simple "objects" that would allow users to compose their libraries in a more object oriented manner. This would open up the possibility of implementing more high-level data structures in an easy way.

There are many other improvements, and features we are considering and hope to implement as time permits. Despite its many warts, we already find Compyle to be remarkably useful.

Conclusions

In this article we have shown how one can implement a two-dimensional molecular dynamics solver using Compyle. The code is parallel from the beginning and runs effortlessly on multi-core CPUs and GPUs without any changes. We have used the example to illustrate the main parallel algorithms that Compyle provides, i.e. elementwise, reduction, and scans. We show how a non-trivial optimization of the example problem is possible using a scan. The results clearly show that we are able to write the code once and have it run on massively parallel architectures. This is very convenient and this is possible because of our approach to the problem which puts parallel algorithms first and forces the user to write code with a hard set of restrictions.

We believe that Compyle allows computational scientists to quickly develop new methods that could benefit from effective parallelization. For molecular dynamics there are many challenges [LGM+15] where this could be useful. While the article used an MD example, and we have ourselves used it in the context of the SPH method [RP+19], Compyle is potentially useful in a variety of other areas. We hope that others are able to use and benefit from using Compyle.

Acknowledgments

We gratefully acknowledge the many open source packages without which this work would never be possible. In particular we thank Andreas Klöckner for many of the parallel algorithms implemented as part of PyOpenCL and PyCUDA that are an inspiration for Compyle. Our thanks to the reviewers for their feedback that has significantly improved the manuscript.

REFERENCES

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56, February 2011. URL: https://doi.org/10.1145/2038037.1941562, doi:10.1145/2038037.1941562.

[LGM+15] Andrea J. Liu, Gary S. Grest, M. Cristina Marchetti, Gregory M. Grason, Mark O. Robbins, Glenn H. Fredrickson, Michael Rubinstein, and Monica Olvera de la Cruz. Opportunities in theoretical and computational polymeric materials and soft matter. *Soft Matter*, 11(12):2326–2332, March 2015. Publisher: The Royal Society of Chemistry. URL: https://pubs.rsc.org/en/content/articlelanding/2015/sm/c4sm02344g, doi:10.1039/C4SM02344G.

[Ram16] Prabhu Ramachandran. PySPH: a reproducible and high-performance framework for smoothed particle hydrodynamics. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 127 – 135, 2016. doi:10.25080/Majora-629e541a-011.

[RP+19] Prabhu Ramachandran, , Kunal Puri, Aditya Bhosale, Dinesh Adepu, Abhinav Muta, Pawan Negi, Rahul Govind, Suraj Sanka, Pankaj Pandey, Chandrashekhar Kaushik, Anshuman Kumar, Ananyo Sen, Rohan Kaushik, Mrinalgouda Patil, Deep Tavker, Dileep Menon, Vikas Kurapati, Amal S Sebastian, Arkopal Dutt, and Arpit Agarwal. PySPH: a Python-based framework for smoothed particle hydrodynamics. *arXiv preprint arXiv:1909.04504*, 2019. URL: https://arxiv.org/abs/1909.04504.

[Sch15] Daniel V. Schroeder. Interactive molecular dynamics. *American Journal of Physics*, 83(3):210–218, February 2015. Publisher: American Association of Physics Teachers. doi:10.1119/1.4901185.

Netlist Analysis and Transformations Using SpyDrNet

Dallin Skouson[‡§*], Andrew Keller[‡§], Michael Wirthlin[‡§]

Abstract—Digital hardware circuits (i.e., for application specific integrated circuits or field programmable gate array circuits) can contain a large number of discrete components and connections. These connections are defined by a data structure called a "netlist". Important information can be gained by analyzing the structure of the circuit netlist and relationships between components. Many specific circuit manipulations require component reorganization in hierarchy and various circuit transformations. SpyDrNet is an open-source netlist analysis and transformation tool written in Python that performs many of these functions. SpyDrNet provides a framework for netlist representation, querying, and modification that is netlist format independent and generalized for use in a wide variety of applications. This tool is actively used to enhance circuit reliability and error detection for circuits operating in harsh radiation environments.

Index Terms—Hardware Design, Netlists, EDA, CAD

Introduction

Digital hardware circuits can contain a large number of discrete components and connections. These components work together through their connections to implement a digital hardware design. Digital hardware circuits are commonly implemented on application specific integrated circuits (ASICs) or on field programmable gate arrays (FPGAs). Discrete components and connections in a digital hardware circuit can be associated with a number of specific attributes. All of this information can be stored inside a graph-like data structure called a "netlist" which details each component and connection along with their respective attributes.

Netlists come in many different formats and organizational structures, but common constructs abound (within EDIF, structural Verliog, and structural VHDL, etc.) [LS89], [JB94]. Most netlist formats have a notion of primitive or basic circuit components that form a basis from which any design can be created. If the contents of a circuit component is unknown, it is treated as a blackbox. Primitive or basic components and blackboxes are viewed as leaf cells. Cells are also referred to as modules, or definitions. Leaf definitions can then be instanced individually inside a larger non-leaf definitions. Definitions and instances contain connection points called pins, which are sometimes grouped together into ports. Nets connect pins together. Nets are also referred to as wires and can be grouped into a collection of nets called a bus or cable.

* *Corresponding author: dallinskouson@byu.edu*
‡ *NSF Center for Space, High-Performance, and Resilient Computing (SHREC)*
§ *Department of Electrical and Computer Engineering, Brigham Young University*

SpyDrNet provides a common framework for representing, querying, and modifying netlists from which application specific analysis and transformation functions can be built. The data structure used to represent netlists is designed to provide quick pointer access to neighboring elements and it is designed to be extensible so that format specific constructs can be stored along with the netlist for preservation when the netlist is exported. This ability supports the representation of a wide variety of netlist formats.

SpyDrNet is currently implemented in Python and provides a Python interface so that it can easily integrate with other Python packages such as NetworkX [HSS08] and PyEDA [CD15]. These library packages have been used in tandem with SpyDrNet to rapidly develop new analysis techniques for better understanding the connectivity and relationships between circuit components as part of reliability research. The Python platform also makes this tool readily available to anyone interested in the community and easily extensible.

This paper presents the SpyDrNet framework, a few use cases, and highlights its use in the development of advanced reliability enhancement techniques. This tool originates from a long line of reliability research focused on improving the reliability of computer circuits implemented on static random access memory based (SRAM-based) FPGAs [JHW+08], [PCC+08], [JW10].

Related Work

The predecessor to SpyDrNet, BYU EDIF Tools [Bri20]. The BYU EDIF tools provide two benefits. First, it provides an API for working with electronic design interchange format (EDIF) netlists. Second, the BYU EDIF Tools includes the Brigham Young University and Los Alamos National Laboratory Triple Modular Redundancy (BL-TMR) Tool. The BL-TMR tool provides a rich set of features for the automated insertion of circuit redundancy for the application of fault-tolerance techniques on digital hardware circuits. These tools have been used extensively in FPGA reliability research [JHW+08], [PCC+08], [JW10].

The BYU EDIF Tools have limitations that motivate the development of SpyDrNet. First, the framework of the BYU EDIF Tools is closely tied to the EDIF netlist format, which makes it challenging to use with alternate netlist formats. Second, the BYU EDIF Tools are primarily intended for use with netlists targeting specific FPGAs. Finally, though not a limitation per se, the BYU EDIF Tools are written in Java and migrating to Python is a motivating factor.

SpyDrNet aims to provide a framework that is netlist format independent and generalized for use in a wide variety of applications. Tools with functionality similar to SpyDrNet exist,

Fig. 1: *The path of a design using SpyDrNet.*

but they tend to be tied to a specific device, architecture, netlist format, or vendor. Some tools with similar functionality such as Vivado [Xil20] or Verific [Ver20] are proprietary. Other tools such as RapidWright [LK18] and Tincr [WN14] are intended for customizing the low-level physical implementation of a netlist on a vendor specific hardware platform. LiveHD [liv] is open-source tool that provides rapid synthesis and simulation updates to small changes in hardware description languages (HDLs). Its framework and language support focuses on the whole design cycle (from logic synthesis, to simulation, to place and route, and tapeout) whereas SpyDrNet focuses specifically on working with structural netlists (i.e., netlists that do not change based on netlist inputs).

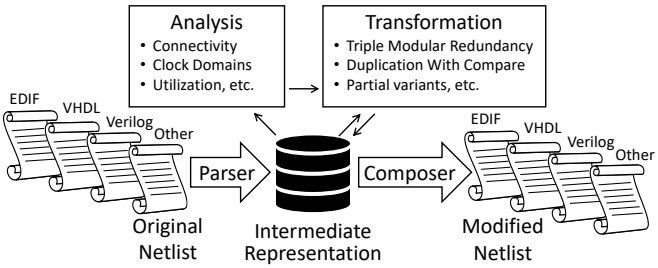


Fig. 2: *Processing a netlist in SpyDrNet. Note that Verilog and VHDL refer to the structural subset of these languages.*

SpyDrNet Tool Flow

Electronic designs may flow through a number of steps before they are built, packaged, or programmed into their target device. For example, these designs may be created in a hardware description language, synthesized into a netlist, then placed, routed, and packaged into a target file which will be used to fabricate the device. A CAD tool can modify the functionality of the final design at any of these stages. The earlier stages in the design flow are slightly less static. Constructs may be optimized out of the design, and the actual hardware implementation of a construct may be unknown. Later in the design process constructs are more stable, but the design is also generally harder to work with (binary files, complex device specific information, etc). By working at the netlist level, SpyDrNet is able to avoid many of the pitfalls of both aspects of the design process.

Figure 1 represents how a design can be prepared and processed prior to and after using SpyDrNet. Many designs start as a hand written hardware description language and are then converted into a netlist using a synthesizer. Netlists are then passed through additional tools to create a design file to be physically implemented.

SpyDrNet currently includes a *parser* and *composer* that imports and exports netlists written in EDIF. Figure 2 shows how the SpyDrNet framework can be used to parse, analyze, transform, and compose netlists in many different formats. Parsers populate an intermediate representation of the netlist in memory using information provided by the input file. With the netlist in intermediate representation, analysis and transformation of the netlist can take place. Once the design is in a state where the user is satisfied, a composer exports the netlist into a desired format. Using the SpyDrNet framework, additional parsers and composers can be written for additional netlist formats.

The Intermediate Representation

The intermediate representation is a generic structural netlist representation employed by SpyDrNet. Structural netlists refer to a class of netlists that represent the interconnection of primative circuit components. These netlists are useful because when modifying netlists for reliability we are less concerned with the general purpose of the circuit and more concerned with how that circuit is implemented. Users can manipulate the structure while in memory and write out a supported format using one of the export modules or *composers* that is included with SpyDrNet. Built into the intermediate representation is an API for manipulating the datastructure.

The data structure was built with a focus on simplifying access to adjacent points in the netlist. In some cases where simple accessors could be added at additional memory cost, the accessors were added. One example of this is the bidirectional references implemented throughout the netlist. This ideology resulted in a slightly longer running time in some cases (and shorter in others), but speed was taken into account as these decisions were made. If a feature significantly increased the run time of the tests, it was examined and optimized.

Primary Data Structures

A short description of some of the data structure components is provided. The constructs behind a structural Netlist are Libraries, Definitions, Instances, Ports, and Cables. Figure 3 shows the connectivity between these components.

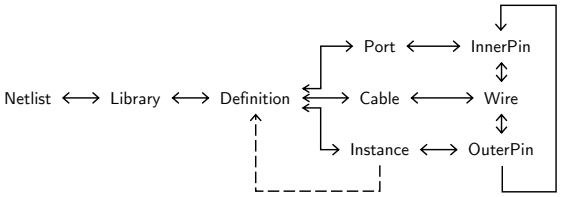


Fig. 3: *Highlights the connectivity between components in the intermediate representation.*

Element: This is the base class for all components of a netlist. Some components are further classified as *first class elements*. First class elements have a name field as well as a properties field.

Definition: These first class elements are sometimes called cells or modules in other representations. They hold all of the information about what their instances contain.

Instance: This first class element is a place holder to be replaced with the sub-elements of the corresponding definition upon build. It is contained in a different definition to its own. In the case of the top level instance it is the place holder that will be replaced by the entire netlist when it is implemented

Port: The Port element can be thought of as containing the information on how a Definition connects the outside world to the elements (instances and cables) it contains.

Cable: Cables are bundles of wires that connect components within a definition. They connect ports to their destination pins.

Pin: These objects represent points of connection between instances or ports and wires. Pins can be divided into inner and outer pin categories. The need for these distinctions lies in the fact that definitions may have more than one instance of itself. Thus components connected on the inside of a definition need to connect to pins related to the definition will connect to inner pins on the definition. Each of these inner pins will correspond to one or more outer pins on instances of the corresponding definition. In this way instances can be connected togehter while still allowing components within a definition to connect to the ports of that definition.

Wire: Wires are grouped inside cables and are elements that help hold connection information between single pins on instances within a definition and within it's ports.

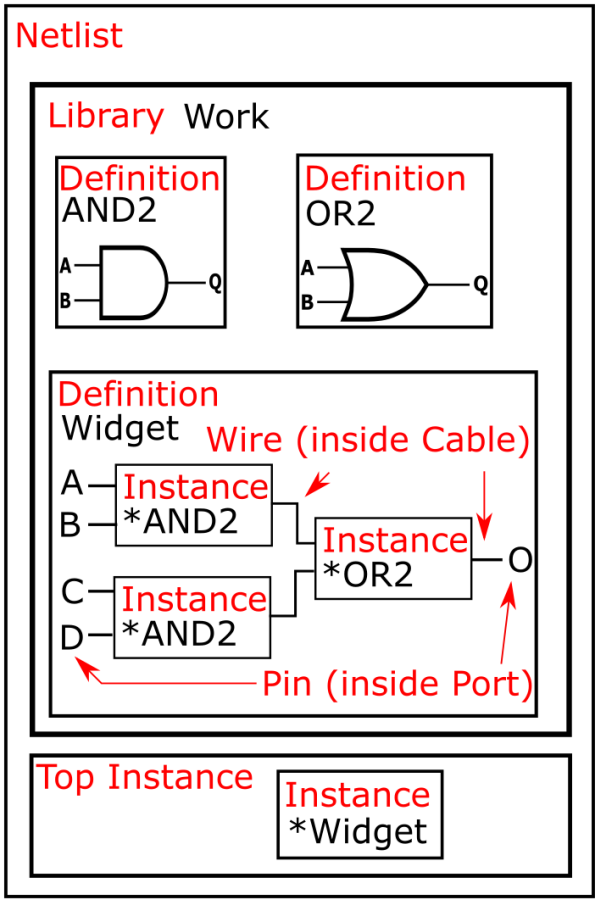


Fig. 4: *Structure of the Intermediate Representation. An asterisk references a definition.*

Support for Multiple Netlist Formats

In addition to holding a generic netlist data structure, the universal netlist representation can hold information specific to individual formats. This is done through the inclusion of metadata dictionaries in many of the SpyDrNet objects.

Parsers can take advantage of the flexibility of the metadata dictionary to carry extra information that source formats present through the tool. This includes information such as comments, parameters, and properties.

In addition, the metadata dictionary can be used to contain any desired user data. Because SpyDrNet is implemented in Python, any data type can be used for the key value in these dictionaries.

Callback Framework

A callback framework was implemented in SpyDrNet to support real time analysis of netlist modifications. Callbacks can assist with applications that make incremental changes to the netlist followed with an analysis of the netlist to determine what more needs to changed. Alternatively users may wish to be warned of violations of design rules such as maintaining unique names. Without callbacks these checks could be performed over the whole netlist data structure on user demand which would add complexity for the end user.

SpyDrNet's callbacks allow users to create plugins that can keep track of the current state of the netlist as changes are made. Currently, a namespace manager is included with SpyDrNet. The callback framework is able to watch changes to the netlist, including addition and removal of elements, as well as changes in naming and structure of the netlist.

Listeners may register to hear these changes as they happen. Each listener is called in the order in which it was registered and may update itself as it sees the netlist change. Plugins that implement listeners can be created and added through the API defined register functions. In general listener functions are expected to receive the same parameters as the function on which they listen.

Modularity Within SpyDrNet

In order to support expansion to a wide variety of netlists, our intermediate representation was designed to reflect a generic netlist data structure. Care was taken to ensure that additional user defined constructs could be easily included in the netlist.

Because of the generic nature of the netlist representation, additional netlist parsers and composers can be built separately and still take full advantage of the existing modification passes available in SpyDrNet. To build a parser or composer requires no more advanced knowledge than an end user may have from using the API to design a custom analysis or modification pass on the netlist.

Other functionality has been added on top of the core of SpyDrNet, including plugin support and the ability to modifiy the netlist at a higher level. These utility functions are used by applications. This layered approach aims to aid in code reusability and reliability allowing lower level functionality to be tested before the higher level functionality is added on.

Analysis and Transformation

SpyDrNet provides a framework for the analysis and transformation of structural netlists. Structural netlists (i.e., a list of circuit components and their connects) capture a hardware design

that is ready for physical implementation where hardware files can be generated (see Figure 1). Information such as component importance or influence can be understood by examining structural relationships between components. Modifications made to the structural netlist are reflected in the hardware implementation.

The analysis and transformation capabilities presented in section form a basis from which custom analysis and transformation functions can be built for specific applications. One current application that benefits from these capabilities is the implementation of duplication with compare (DWC) and triple modular redundancy (TMR) to circuit designs, which is discussed later on. Using SpyDrNet's analysis and transformations allows end-users to rapidly develop custom functions for specific needs.

Utility Functions

SpyDrNet has several high level features currently included. All of these features have an impact on the overall netlist structure but several are most useful when included in other applications. This section will highlight some of the simpler high level features that are currently implemented in SpyDrNet.

Basic Functionality

Functionality is provided through the API to allow for creation and modification of elements in the netlist data structures. Sufficient functionality is provided to create a netlist from the ground up, and read all available information from a created netlist. Netlist objects are completely mutable and allow for on demand modification. This provides a flexible framework upon which users can build and edit netlists data structures. The basic functionality includes functionality to create new children elements, modify the properties of elements, delete elements, and change the relationships of elements. All references bidirectional and otherwise are maintained behind the scenes to ensure the user can easily complete modification passes on the netlist while maintaining a valid representation.

The mutability of the objects in SpyDrNet is of special mention. Many frameworks require that the object's name be set on creation, and disallow any changes to that name. SpyDrNet, on the other hand, allows name changes as well as any other changes to the connections, and properties of the objects. The callback framework, as discussed in another section, provides hooks that allow checks for violations of user defined rules if desired.

Examples of some of the basic functionality are highlighted in the following code segment. Relationships, such as the reference member of the instances and the children of these references are members of the SpyDrNet objects. Additional key data can be accessed as members of the classes. Other format specific data can be accessed through dictionary lookups. Since the name is also key data but, is not required it can be looked up through either access method as noted in one of the single line comment.

```
import spydrnet as sdn

netlist = sdn.load_example_netlist_by_name(
    'fourBitCounter')

top_instance = netlist.top_instance

def recurse(instance, depth):
    '''print something like this:
    top
        child1
            child1.child
        child2
            child2.child'''
    s = depth * "\t"

    #instance.name could also be instance["NAME"]
    print(
        s, instance.name,
        "(", instance.reference.name, ")")
    for c in instance.reference.children:
        recurse(c, depth + 1)

recurse(top_instance, 0)
```

Hierarchy

Netlists can be hierarchical or they can be flat (see Figure 5). Hierarchical netlists contain non-leaf instances, which instance a definition that contains additional instances. Flat netlists contain only leaf instances, which instance a definition that is void of additional instances. SpyDrNet supports hierarchy and performing analysis and transformations across hierarchical boundaries. SpyDrNet focuses on structural netlists that are static (i.e., netlists that do not change based on inputs to the netlist).

```
Top_Inst (Top_Def)
├ Mid_Inst_A (Mid_Def)
│  ├ Leaf_Inst_A (Leaf_Def)
│  ├ Leaf_Inst_B (Leaf_Def)
│  └ Leaf_Inst_C (Leaf_Def)
├ Mid_Inst_B (Mid_Def)
│  ├ Leaf_Inst_A (Leaf_Def)
│  ├ Leaf_Inst_B (Leaf_Def)
│  └ Leaf_Inst_C (Leaf_Def)
└ Mid_Inst_C (Mid_Def)
   ├ Leaf_Inst_A (Leaf_Def)
   ├ Leaf_Inst_B (Leaf_Def)
   └ Leaf_Inst_C (Leaf_Def)
```

```
Top_Inst (Top_Def)
├ Mid_Inst_A/Leaf_Inst_A (Leaf_Def)
├ Mid_Inst_A/Leaf_Inst_B (Leaf_Def)
├ Mid_Inst_A/Leaf_Inst_C (Leaf_Def)
├ Mid_Inst_B/Leaf_Inst_A (Leaf_Def)
├ Mid_Inst_B/Leaf_Inst_B (Leaf_Def)
├ Mid_Inst_B/Leaf_Inst_C (Leaf_Def)
├ Mid_Inst_C/Leaf_Inst_A (Leaf_Def)
├ Mid_Inst_C/Leaf_Inst_B (Leaf_Def)
└ Mid_Inst_C/Leaf_Inst_C (Leaf_Def)
```

Fig. 5: *A hierarchical netlist (left) versus a flat netlist (right).*

Hierarchy is by default a component of many netlist formats. One of the main advantages to including hierarchy in a design is the ability to abstract away some of the finer details on a level based system, while still including all of the information needed to build the design. The design's hierarchical information is maintained in SpyDrNet by having definitions instanced within other definitions.

SpyDrNet allows the user to work with the structure of a netlist directly, having only one of each instance per hierarchical level, but it also allows the user view the netlist instances in a hierarchical context through the use of hierarchical references as outlined below. Some other tools only provide the hierarchical representation of the design.

There are drawbacks and advantages to each view on the netlist, but the inclusion of a hierarchical view helps allow users to make the fewest possible unneeded changes to the design. Additionally there are several advantages to maintaining hierarchy, smaller file sizes are possible in some cases, as sub components do not need to be replicated. Simulators may have an easier time predicting how the design will act once implemented [DIR+04]. Further research could be done to analyze the impact of hierarchy on later compilation steps.

Flattening

SpyDrNet has the ability to flatten hierarchical designs. One method to remove hierarchy from a design is to move all of the sub components to the top level of the netlist repeatedly until each sub component at the top level is a terminal instance, where no more structural information is included below that instance's level.

Flattening was added to SpyDrNet because there are some algorithms which can be applied more simply on a flat design. Algorithms in which a flat design may be simpler to work with are graph analysis, and other algorithms where the connections between low level components are of interest.

Included is an example of how one might flatten a netlist in SpyDrNet.

```
import spydrnet as sdn
from sdn.flatten import flatten

netlist = sdn.load_example_netlist_by_name(
    'fourBitCounter')

#flattens in place. netlist will now be flat.
flatten(netlist)
```

Uniquify

Uniquify ensures that each non-terminal instance is unique, meaning that it and it's definition have a one to one relationship. Non-unique definitions and instances may exist in most netlist formats. One such example could be a four bit adder that is composed of four single bit adders. Assuming that each single bit adder is composed of more than just a single component on the target device, and that the single bit adders are all identical, the design may just define a single single bit adder which it uses in four places. To uniquify this design, new matching definitions for single bit adders would be created for each of the instances of the original single bit adder and the instances that correspond would be pointed to the new copied definitions. Thus each of the definitions would be left with a single instance.

The uniquify algorithm is very useful when modifications are desired on a specific part of the netlist but not to all instances of the particular component. For example in the four bit adder, highlighted in the previous paragraph of this section, if we assume that the highest bit does not need a carry out, the single bit adder there could be simplified. However, if we make modifications to the single bit adder before uniquifying the modifications will apply to all four adders. If we instead uniquify first then we can easily modify only the adder of interest.

Currently `Uniquify` is implemented to ensure that the entire netlist contains only unique definitions. This is one approach to uniquify, however an interesting area for future exploration is that of uniquify on demand. Or some other approach to only ensure and correct uniquification of modified components only. This is left for future work.

The following code example shows uniquify being used in SpyDrNet.

```
import spydrnet as sdn
from sdn.uniquify import uniquify

netlist = sdn.load_example_netlist_by_name(
    'fourBitCounter')

uniquify(netlist)
```

Clone

Cloning is another useful algorithm currently implemented in SpyDrNet. Currently all of the components in a netlist can be cloned from pins and wires to whole netlist objects. Upon initial inspection clone seems simple. However, there is some complexity when it comes to the connections between individual components. Some explanation is provided here.

Clone could be implemented a number of ways. We attempted to find the logical method for our clone algorithm at each level of the data structure. Our overall guiding principles were that at each level, lower level objects should maintain their connections, the cloned object should not belong to any other object, and the cloned object should not maintain its horizontal connections. There are of course some exceptions to these rules which seemed judicious. One such example is that when cloning an instance, That instance will maintain its original corresponding definition, unless the corresponding definition is also being cloned as in the case of cloning a whole library or netlist (in which case the new cloned definition will be used).

Additionally connection modification was done at a level lower than the API in order to maintain consistency as different components were cloned. This promoted code reuse in the clone implementation and helped minimize the number of dictionaries used.

The clone algorithm is very useful while implementing some of the higher level algorithms such as TMR and DWC with compare that we use for reliability research. In these algorithms cloning is essential, and having it built into the tool helps simplify their implementation.

The example code included in this section will clone an element and then add that element back into the netlist which it originally belonged to. Comments are included for most lines in this example to illuminate why each step must be taken.

```
import spydrnet as sdn

netlist = sdn.load_example_netlist_by_name(
    'hierarchical_luts')

#index found by printing children's names
sub = netlist.top_instance.reference.children[2]
sub_clone =
    sub.clone()

#renamed needed to be added back into the netlist
sub_clone.name = "sub_clone"

#this line adds the cloned instance into the netlist
netlist.top_instance.reference.add_child(sub_clone)
```

Hierarchical References

SpyDrNet includes the ability to create a hierarchical reference graph of all of the instances, ports, cables, and other objects which may be instantiated. The goal behind hierarchical references is to create a graph on which other tools, such as NetworkX can more easily build a graph. each hierarchical reference will be unique, even if the underlying component is not unique. These components are also very light weight to minimize memory impact since there can be many of these in flight at one time.

The code below shows how one can get and print hierarchical references. The hierarchical references can represent any spydrnet object that may be instantiated in a hierarchical manner.

```
top = netlist.top_instance
child_instances = top.reference.children

for h in sdn.get_hinstances(child_instances):
    print(h, type(h.item).__name__)
```

Getter Functions

SpyDrNet includes getter functions which are helpful in the analysis and transformation of netlists. These functions were created

to help a user more quickly traverse the netlist. These functions provide the user with quick access to adjacent components. A call to a getter function can get any other related elements from the existing element that the user has a handle to (see Figure 6). Similar to clone there are multiple methods which could be used to implement a correct getter function. We again strove to apply the most logical and consistent rules for the getter functions. There are some places in which the object returned may not be the only possible object to be returned. In these cases generators are returned. In cases in which there are two possible classes of relationships upon which to return objects, the user may specify whether they would like to get the more inward related or outward related objects. For example, a port may have outer pins on instances or inner pins within the port in the definition. Both of these pins can be obtained separately by passing a flag.

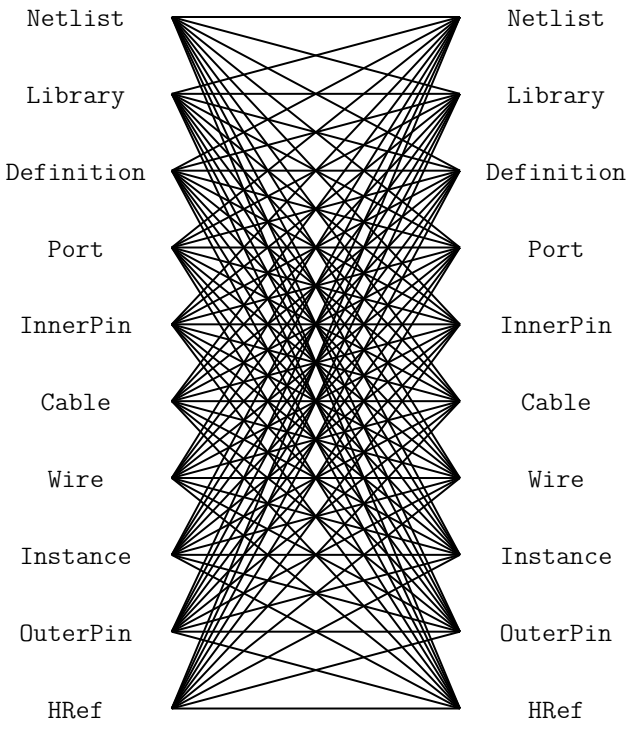


Fig. 6: *Getter functions are able to get sets of any element related to any other element.*

In the example only a few of the possible getter functions are shown. The same pattern can be used to get any type of object from another however. Each call to a getter function returns a generator.

Example Applications

SpyDrNet may be used for a wide variety of applications. SpyDrNet grew out of a lab that is focused primarily on improving circuit reliability and security. An application that has had strong influence over its development is that of enhancing circuit reliability in harsh radiation environments through partial circuit replication [PCC+08]. When a particle of ionizing radiation passes through an integrated circuit, it can deposit enough energy to invert values stored in memory cells [JED06]. An FPGA is a computer chip that can be used to implement custom circuits. SRAM-based FPGA stores a circuits configuration in a large array of memory. When radiation corrupts an FPGA configuration memory, it can corrupt the underlying circuit and cause failure.

One of our areas of research involves finding ways to design more reliable circuits to be programmed onto existing, non specialized, FPGAs. These modifications are useful for designers that deploy many FPGAs as well as designers that plan on deploying circuits in high radiation environments where single event upsets can disrupt the normal operation of devices. These reliability focused modifications require some analysis of netlist structure as well as modifications in the netlist.

SpyDrNet was created to help automate this process and allow our researchers to spend more time studying the resulting improved circuitry and less time modifying the circuit itself. It is important to note that some care needs to be taken to ensure that redundancy modifications are not removed by down stream optimizations in implementation. Reliability modifications to netlists are often optimized away. One common adjustment to a netlist for reliability purposes, is a replication of various components. Often when tools see the same functionality with a theoretical identical result they will attempt to remove the duplicated portion and provide two outputs on a single instance. This defeats the purpose of the reliability modifications. Using and modifying netlists allows us to bypass those optimizations and gives more control over how our design is built. Below are some details on using SpyDrNet for higher level transformation and analysis techniques applicable to reliability applications.

Triple Modular Redundancy

TMR is one method by which circuits can be made more reliable. TMR triplicates portions of the circuit to allow the circuit to continue to provide the correct result even under some cases of error. Voters are inserted between triplicated circuit components to pass the most common result on to the next stage of the circuit [PCC+08]. Figure 7 shows two typical layouts for TMR. The top half of the image shows a triplicated circuit with a single voter that feeds into the next stage of the circuit. The bottom of the figure shows a triplicated voter layout such that even a single voter failure may be tolerated.

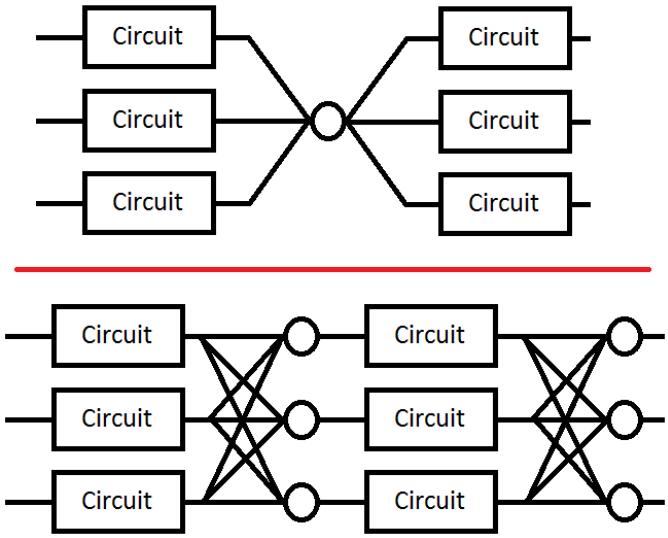


Fig. 7: *Triple modular redundancy with a single voter and triplicated voters. [tmr]*

TMR has been applied using SpyDrNet. The current implementation selects subsets of the circuit to replicate. Then a voter insertion algorithm creates and inserts the voter logic between triplicated layers. Later, reduction voting is added to the output, connecting the triplicated logic in place of the original implementation. The ability of SpyDrNet to carry hierarchy through the tool was taken advantage of by the TMR implementation. This allows the triplicated design to take advantage of the benefits of hierarchy including, improved place and route steps on the target FPGA. Previous work with the BYU EDIF Tools [Bri20] required a flattened design to accomplish TMR on a netlist. The triplicated design was programmed to an FPGA after being processed using SpyDrNet.

Duplication With Compare

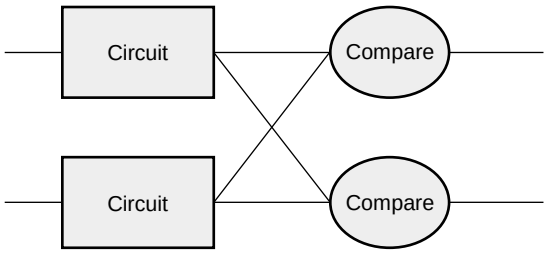


Fig. 8: *Duplication with compare showing the duplicated circuitry and duplicated violation flags.*

DWC is a reliability algorithm in which the user will duplicate components of the design and include comparators on the output to try present a flag that will be raised when one of the circuits goes down [JHW+08]. Like TMR's voters, the comparators can be duplicated as well to ensure that if a comparator goes down at least one of the comparators will flag an issue.

DWC was again implemented on SpyDrNet. Once again this was able to take advantage of SpyDrNet's hierarchy and maintain that through the build. Comparators were created and inserted and the selected portion of the design was duplicated. The resulting circuits were programmed to an FPGA after being read into SpyDrNet, modified and written back out. As with TMR the existing implementation on the BYU EDIF Tools [Bri20] required that the design be flattened before being processed.

Clock Domain Analysis

In hardware various clocks are often used in different portions of the circuit. Sometimes inputs and outputs will come in on a different clock before they reach the main pipeline of the circuit. At the junctions between clock domains circutry should not be triplicated in TMR. If it is triplicated it may result in steady state error on the output because the signals from the three inputs may reach the crossing at different times and be registered improperly [LNW10]. This can make the overall reliability of the system lower than it otherwise would be.

In order to find these locations. Clock domains have been examined using SpyDrNet. The basic methodology for doing this was to find the clock ports on the various components in the design which have them and trace those clocks through the netlist. The resulting connected components form a clock domain. When a triplication pass encountered the boundry between domains the triplicated circuit could be reduced to a single signal to cross the boundry.

Graph Analysis and Feedback

While triplictaing a design users must determine the best location to insert voters in the design. Voters could be inserted liberally at the cost of the timing of the critical path. Alternatively sparse voter insertion can yield a lower reliability. One consideration to take into account is that voters inserted on feedback loops in the directional graph represented by the netlist can help correct the circuit's state more readily. One study concluded that inserting voters after high fanout flip flops in a design yielded good results. [JW10] This voter insertion algorithm was implemented on SpyDrNet after doing analysis using NetworkX [HSS08] to find the feedback loops.

Future Direction

As SpyDrNet matures, several new features are planned to benefit SpyDrNet's users. Several of the upcoming features are discussed here but a more complete roadmap is maintained with the project's repository.

Additional netlist format parsers and composers are planned. Supplying additional parser and composers will open the door for users to more easily use SpyDrNet with a wider variety of technologies and device vendor tools. This work will enable conversion between formats as well, which will provide greater flexibility for end users. Some vendor tools only accept specific netlist formats. Converting netlist formats would provide further possibilities.

Plans to integrate more closely with other open source tools in analysis and hardware design have been made. These plans include further work to ensure NetworkX and other SciPy utilities can be easily leveraged by SpyDrNet. Integrating with additional open source electionic design tools is also of interest, which could help make SpyDrNet a useful part of an open source design flow.

SpyDrNet was designed to be generic and modular to allow for support of a wide variety of netlist formats. Device specific information is not included in SpyDrNet. Future work may include providing a framwork to maintain and make use of device specific data. Such a framework could simplify a number of different applications that require device specific information. Device data of interest may include device resource constraints, clock propagation behavior, and limitations on how components can be implemented on a specific technology. Providing users a simpler way of maintaining and utilizing this data will help improve the flexibility of the tool.

Several portions of SpyDrNet could be sped up by accelerating them in C/C++. Parseing netlists can take several minutes for very large designs using the current implementation. An accelerated verion of the current parser would be of use in the future as more users with increasingly complex designs become interested in SpyDrNet.

Conclusion

SpyDrNet is a framework created to be as flexible as possible while still meeting the needs of reliability related research. We have worked to ensure that this tool is capable of a wide variety of netlist modifications.

Although this tool is new, a few reliability applications have been built on SpyDrNet. Because of these applications we feel confident that this tool can be helpful to others. SpyDrNet is released on github under an open source licence. New users are welcome to use and contribute to the SpyDrNet tools.

Acknowledgment

This work was supported by the Utah NASA Space Grant Consortium and by the I/UCRC Program of the National Science Foundation under Grant No. 1738550.

REFERENCES

[Bri20] Brigham Young University. BYU EDIF Tools [online]. 2020. URL: https://sourceforge.net/projects/byuediftools/.

[CD15] Chris Drake. PyEDA: Data Structures and Algorithms for Electronic Design Automation. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 25 – 30, 2015. doi:10.25080/Majora-7b98e3ed-004.

[DIR+04] P. Daglio, D. Iezzi, D. Rimondi, C. Roma, and S. Santapa. Building the hierarchy from a flat netlist for a fast and accurate post-layout simulation with parasitic components. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 336–337 Vol.3, Feb 2004. doi:10.1109/DATE.2004.1269268.

[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.

[JB94] Jen-Jen Lung and J. Bhasker. Verilog netlist as an exchange language. In *International Verilog HDL Conference*, pages 10–14, March 1994. doi:10.1109/IVC.1994.323754.

[JED06] Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices, 2006. URL: https://www.jedec.org/sites/default/files/docs/JESD89A.pdf.

[JHW+08] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan. Using duplication with compare for on-line error detection in fpga-based designs. In *2008 IEEE Aerospace Conference*, pages 1–11, March 2008. doi:10.1109/AERO.2008.4526470.

[JW10] Jonathan M Johnson and Michael Wirthlin. Voter Insertion Algorithms for {FPGA} Designs Using Triple Modular Redundancy. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 249–258, New York, NY, USA, 2010. ACM. doi:10.1145/1723112.1723154.

[liv] LiveHD: Live hardware development. https://github.com/masc-ucsc/livehd.

[LK18] C. Lavin and A. Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140, April 2018. doi:10.1109/FCCM.2018.00030.

[LNW10] Y. Li, B. Nelson, and M. Wirthlin. Synchronization techniques for crossing multiple clock domains in fpga-based tmr circuits. *IEEE Transactions on Nuclear Science*, 57(6):3506–3514, Dec 2010. doi:10.1109/TNS.2010.2086075.

[LS89] W. Li and H. Switzer. A unified data exchnage environment based on edif. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, DAC '89, page 803–806, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/74382.74534.

[PCC+08] Brian Pratt, Michael Caffrey, James F Carroll, Paul Graham, Keith Morgan, and Michael Wirthlin. Fine-grain SEU mitigation for FPGAs using partial TMR. *IEEE Transactions on Nuclear Science*, 55(4):2274–2280, aug 2008. doi:10.1109/TNS.2008.2000852.

[tmr] Graphical Representation of TMR. Mewtow / CC BY-SA (https://creativecommons.org/licenses/by-sa/4.0). URL: https://commons.wikimedia.org/wiki/File:Triple_Modular_Redundancy_et_sa_variante_am%C3%A9lior%C3%A9e.png.

[Ver20] Verific Design Automation, Inc. Verific Design Automation [online]. 2020. URL: https://www.verific.com/.

[WN14] B. White and B. Nelson. Tincr — a custom cad tool framework for vivado. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014. doi:10.1109/ReConFig.2014.7032560.

[Xil20] Xilinx, Inc. Vivado Design Suite [online]. 2020. URL: https://www.xilinx.com/products/design-tools/vivado.html.

Introduction to Geometric Learning in Python with Geomstats

Nina Miolane[‡*], Nicolas Guigui[§], Hadi Zaatiti, Christian Shewmake, Hatem Hajri, Daniel Brooks, Alice Le Brigant, Johan Mathe, Benjamin Hou, Yann Thanwerdas, Stefan Heyder, Olivier Peltre, Niklas Koep, Yann Cabanes, Thomas Gerald, Paul Chauchat, Bernhard Kainz, Claire Donnat, Susan Holmes, Xavier Pennec

https://youtu.be/Ju-Wsd84uG0

Abstract—There is a growing interest in leveraging differential geometry in the machine learning community. Yet, the adoption of the associated geometric computations has been inhibited by the lack of a reference implementation. Such an implementation should typically allow its users: (i) to get intuition on concepts from differential geometry through a hands-on approach, often not provided by traditional textbooks; and (ii) to run geometric machine learning algorithms seamlessly, without delving into the mathematical details. To address this gap, we present the open-source Python package `geomstats` and introduce hands-on tutorials for differential geometry and geometric machine learning algorithms - Geometric Learning - that rely on it. Code and documentation: `github.com/geomstats/geomstats` and `geomstats.ai`.

Index Terms—differential geometry, statistics, manifold, machine learning

Introduction

Data on manifolds arise naturally in different fields. Hyperspheres model directional data in molecular and protein biology [KH05] and some aspects of 3D shapes [JDM12], [HVS+16]. Density estimation on hyperbolic spaces arises to model electrical impedances [HKKM10], networks [AS14], or reflection coefficients extracted from a radar signal [CBA15]. Symmetric Positive Definite (SPD) matrices are used to characterize data from Diffusion Tensor Imaging (DTI) [PFA06], [YZLM12] and functional Magnetic Resonance Imaging (fMRI) [STK05]. These manifolds are curved, differentiable generalizations of vector spaces. Learning from data on manifolds thus requires techniques from the mathematical discipline of differential geometry. As a result, there is a growing interest in leveraging differential geometry in the machine learning community, supported by the fields of Geometric Learning and Geometric Deep Learning [BBL+17].

Despite this need, the adoption of differential geometric computations has been inhibited by the lack of a reference implementation. Projects implementing code for geometric tools are often custom-built for specific problems and are not easily reused. Some Python packages do exist, but they mainly focus on optimization (Pymanopt [TKW16], Geoopt [BG18], [Koc19], McTorch [MJK+18]), are dedicated to a single manifold (PyRiemann [Bar15], PyQuaternion [Wyn14], PyGeometry [Cen12]), or lack unit-tests and continuous integration (TheanoGeometry [KS17]). An open-source, low-level implementation of differential geometry and associated learning algorithms for manifold-valued data is thus thoroughly welcome.

`Geomstats` is an open-source Python package built for machine learning with data on non-linear manifolds [MGLB+]: a field called Geometric Learning. The library provides object-oriented and extensively unit-tested implementations of essential manifolds, operations, and learning methods with support for different execution backends - namely NumPy, PyTorch, and TensorFlow. This paper illustrates the use of `geomstats` through hands-on introductory tutorials of Geometric Learning. These tutorials enable users: (i) to build intuition for differential geometry through a hands-on approach, often not provided by traditional textbooks; and (ii) to run geometric machine learning algorithms seamlessly without delving into the lower-level computational or mathematical details. We emphasize that the tutorials are not meant to replace theoretical expositions of differential geometry and geometric learning [Pos01], [PSF19]. Rather, they will complement them with an intuitive, didactic, and engineering-oriented approach.

Presentation of Geomstats

The package geomstats is organized into two main modules: geometry and learning. The module `geometry` implements low-level differential geometry with an object-oriented paradigm and two main parent classes: `Manifold` and `RiemannianMetric`. Standard manifolds like the `Hypersphere` or the `Hyperbolic` space are classes that inherit from `Manifold`. At the time of writing, there are over 15 manifolds implemented in `geomstats`. The class `RiemannianMetric` provides computations related to Riemannian geometry on such manifolds such as the inner product of two tangent vectors at a base point, the geodesic distance between two points, the Exponential and Logarithm maps at a base point, and many others.

The module `learning` implements statistics and machine learning algorithms for data on manifolds. The code is object-oriented and classes inherit from `scikit-learn` base classes and mixins such as `BaseEstimator`, `ClassifierMixin`, or `RegressorMixin`. This module provides implementations

* *Corresponding author: nmiolane@stanford.edu*
‡ *Stanford University*
§ *Université Côte d'Azur, Inria*

of Fréchet mean estimators, K-means, and principal component analysis (PCA) designed for manifold data. The algorithms can be applied seamlessly to the different manifolds implemented in the library.

The code follows international standards for readability and ease of collaboration, is vectorized for batch computations, undergoes unit-testing with continuous integration, and incorporates both TensorFlow and PyTorch backends to allow for GPU acceleration. The package comes with a visualization module that enables users to visualize and further develop an intuition for differential geometry. In addition, the datasets module provides instructive toy datasets on manifolds. The repositories examples and notebooks provide convenient starting points to get familiar with `geomstats`.

First Steps

To begin, we need to install `geomstats`. We follow the installation procedure described in the first steps of the online documentation. Next, in the command line, we choose the backend of interest: NumPy, PyTorch or TensorFlow. Then, we open the iPython notebook and import the backend together with the visualization module. In the command line:

```
export GEOMSTATS_BACKEND=numpy
```

then, in the notebook:

```
import geomstats.backend as gs
import geomstats.visualization as visualization

visualization.tutorial_matplotlib()
```

```
INFO: Using numpy backend
```

Modules related to `matplotlib` and `logging` should be imported during setup too. More details on setup can be found on the documentation website: `geomstats.ai`. All standard NumPy functions should be called using the `gs.` prefix - e.g. `gs.exp`, `gs.log` - in order to automatically use the backend of interest.

Tutorial: Statistics and Geometric Statistics

This tutorial illustrates how Geometric Statistics and Learning differ from traditional Statistics. Statistical theory is usually defined for data belonging to vector spaces, which are linear spaces. For example, we know how to compute the mean of a set of numbers or of multidimensional arrays.

Now consider a non-linear space: a manifold. A manifold M of dimension m is a space that is possibly curved but that looks like an m-dimensional vector space in a small neighborhood of every point. A sphere, like the earth, is a good example of a manifold. What happens when we apply statistical theory defined for linear vector spaces to data that does not naturally belong to a linear space? For example, what happens if we want to perform statistics on the coordinates of world cities lying on the earth's surface: a sphere? Let us compute the mean of two data points on the sphere using the traditional definition of the mean.

```
from geomstats.geometry.hypersphere import \
    Hypersphere

n_samples = 2
sphere = Hypersphere(dim=2)
points_in_manifold = sphere.random_uniform(
     n_samples=n_samples)
```

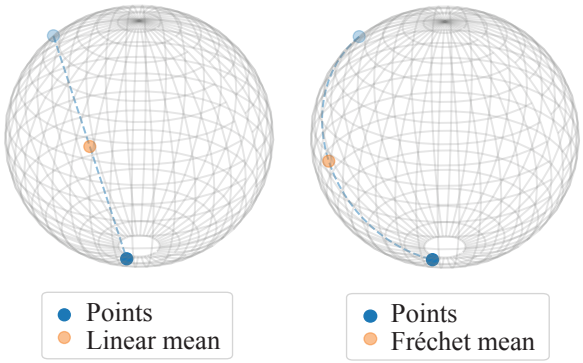


Fig. 1: *Left: Linear mean of two points on the sphere. Right: Fréchet mean of two points on the sphere. The linear mean does not belong to the sphere, while the Fréchet mean does. This illustrates how linear statistics can be generalized to data on manifolds, such as points on the sphere.*

```
linear_mean = gs.sum(
    points_in_manifold, axis=0) / n_samples
```

The result is shown in Figure 1 (left). What happened? The mean of two points on a manifold (the sphere) is not on the manifold. In our example, the mean of these cities is not on the earth's surface. This leads to errors in statistical computations. The line `sphere.belongs(linear_mean)` returns `False`. For this reason, researchers aim to build a theory of statistics that is - by construction - compatible with any structure with which we equip the manifold. This theory is called Geometric Statistics, and the associated learning algorithms: Geometric Learning.

In this specific example of mean computation, Geometric Statistics provides a generalization of the definition of "mean" to manifolds: the Fréchet mean.

```
from geomstats.learning.frechet_mean import \
    FrechetMean

estimator = FrechetMean(metric=sphere.metric)
estimator.fit(points_in_manifold)
frechet_mean = estimator.estimate_
```

Notice in this code snippet that `geomstats` provides classes and methods whose API will be instantly familiar to users of the widely-adopted `scikit-learn`. We plot the result in Figure 1 (right). Observe that the Fréchet mean now belongs to the surface of the sphere!

Beyond the computation of the mean, `geomstats` provides statistics and learning algorithms on manifolds that leverage their specific geometric structure. Such algorithms rely on elementary operations that are introduced in the next tutorial.

Tutorial: Elementary Operations for Data on Manifolds

The previous tutorial showed why we need to generalize traditional statistics for data on manifolds. This tutorial shows how to perform the elementary operations that allow us to "translate" learning algorithms from linear spaces to manifolds.

We import data that lie on a manifold: the world cities dataset, that contains coordinates of cities on the earth's surface. We visualize it in Figure 2.

```
import geomstats.datasets.utils as data_utils

data, names = data_utils.load_cities()
```

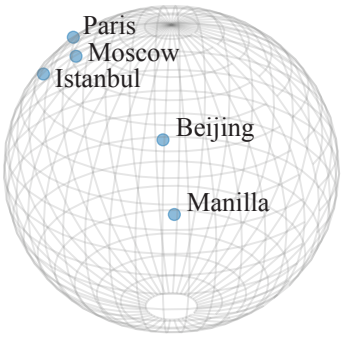


Fig. 2: *Subset of the world cities dataset, available in* `geomstats` *with the function* `load_cities` *from the module* `datasets.utils`*. Cities' coordinates are data on the sphere, which is an example of a manifold.*

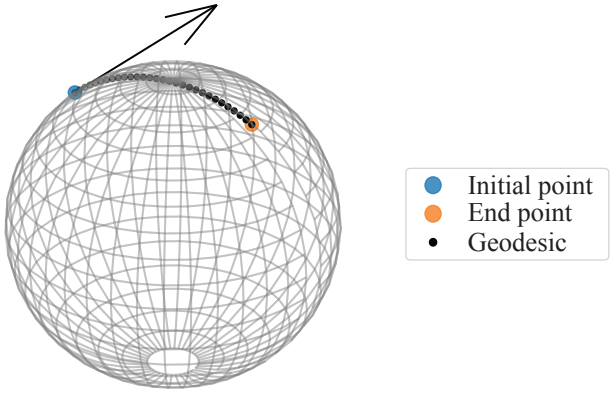


Fig. 3: *Elementary operations on manifolds illustrated on the sphere. The exponential map at the initial point (blue point) shoots the black tangent vector along the geodesic, and gives the end point (orange point). Conversely, the logarithm map at the initial point (blue point) takes the end point (orange point) as input, and outputs the black tangent vector. The geodesic between the blue point and the orange point represents the path of shortest length between the two points.*

How can we compute with data that lie on such a manifold? The elementary operations on a vector space are addition and subtraction. In a vector space (in fact seen as an affine space), we can add a vector to a point and subtract two points to get a vector. Can we generalize these operations in order to compute on manifolds?

For points on a manifold, such as the sphere, the same operations are not permitted. Indeed, adding a vector to a point will not give a point that belongs to the manifold: in Figure 3, adding the black tangent vector to the blue point gives a point that is outside the surface of the sphere. So, we need to generalize to manifolds the operations of addition and subtraction.

On manifolds, the exponential map is the operation that generalizes the addition of a vector to a point. The exponential map takes the following inputs: a point and a tangent vector to the manifold at that point. These are shown in Figure 3 using the blue point and its tangent vector, respectively. The exponential map returns the point on the manifold that is reached by "shooting" with the tangent vector from the point. "Shooting" means following a "geodesic" on the manifold, which is the dotted path in Figure 3. A geodesic, roughly, is the analog of a straight line for general manifolds - the path whose, length, or energy, is minimal between two points, where the notions of length and energy are defined by the Riemannian metric. This code snippet shows how to compute the exponential map and the geodesic with `geomstats`.

```
from geomstats.geometry.hypersphere import \
    Hypersphere

sphere = Hypersphere(dim=2)

initial_point = paris = data[19]
vector = gs.array([1, 0, 0.8])
tangent_vector = sphere.to_tangent(
     vector, base_point=initial_point)

end_point = sphere.metric.exp(
    tangent_vector, base_point=initial_point)

geodesic = sphere.metric.geodesic(
    initial_point=initial_point,
    initial_tangent_vec=tangent_vector)
```

Similarly, on manifolds, the logarithm map is the operation that generalizes the subtraction of two points on vector spaces. The logarithm map takes two points on the manifold as inputs and returns the tangent vector required to "shoot" from one point to the other. At any point, it is the inverse of the exponential map. In Figure 3, the logarithm of the orange point at the blue point returns the tangent vector in black. This code snippet shows how to compute the logarithm map with `geomstats`.

```
log = sphere.metric.log(
    point=end_point, base_point=initial_point)
```

We emphasize that the exponential and logarithm maps depend on the "Riemannian metric" chosen for a given manifold: observe in the code snippets that they are not methods of the `sphere` object, but rather of its `metric` attribute. The Riemannian metric defines the notion of exponential, logarithm, geodesic and distance between points on the manifold. We could have chosen a different metric on the sphere that would have changed the distance between the points: with a different metric, the "sphere" could, for example, look like an ellipsoid.

Using the exponential and logarithm maps instead of linear addition and subtraction, many learning algorithms can be generalized to manifolds. We illustrated the use of the exponential and logarithm maps on the sphere only; yet, `geomstats` provides their implementation for over 15 different manifolds in its `geometry` module with support for a variety of Riemannian metrics. Consequently, `geomstats` also implements learning algorithms on manifolds, taking into account their specific geometric structure by relying on the operations we just introduced. The next tutorials show more involved examples of such geometric learning algorithms.

Tutorial: Classification of SPD Matrices

Tutorial context and description

We demonstrate that any standard machine learning algorithm can be applied to data on manifolds while respecting their geometry. In the previous tutorials, we saw that linear operations (mean, linear weighting, addition and subtraction) are not defined on manifolds. However, each point on a manifold has an associated tangent space which is a vector space. As such, in the tangent space, these operations are well defined! Therefore, we can use the logarithm map (see Figure 3 from the previous tutorial) to go from points on

manifolds to vectors in the tangent space at a reference point. This first strategy enables the use of traditional learning algorithms on manifolds.

A second strategy can be designed for learning algorithms, such as K-Nearest Neighbors classification, that rely only on distances or dissimilarity metrics. In this case, we can compute the pairwise distances between the data points on the manifold, using the method `metric.dist`, and feed them to the chosen algorithm.

Both strategies can be applied to any manifold-valued data. In this tutorial, we consider symmetric positive definite (SPD) matrices from brain connectomics data and perform logistic regression and K-Nearest Neighbors classification.

SPD matrices in the literature

Before diving into the tutorial, let us recall a few applications of SPD matrices in the machine learning literature. SPD matrices are ubiquitous across many fields [CS16], either as input of or output to a given problem. In DTI for instance, voxels are represented by "diffusion tensors" which are 3x3 SPD matrices representing ellipsoids in their structure. These ellipsoids spatially characterize the diffusion of water molecules in various tissues. Each DTI thus consists of a field of SPD matrices, where each point in space corresponds to an SPD matrix. These matrices then serve as inputs to regression models. In [YZLM12] for example, the authors use an intrinsic local polynomial regression to compare fiber tracts between HIV subjects and a control group. Similarly, in fMRI, it is possible to extract connectivity graphs from time series of patients' resting-state images [WZD+13]. The regularized graph Laplacians of these graphs form a dataset of SPD matrices. This provides a compact summary of brain connectivity patterns which is useful for assessing neurological responses to a variety of stimuli, such as drugs or patient's activities.

More generally speaking, covariance matrices are also SPD matrices which appear in many settings. Covariance clustering can be used for various applications such as sound compression in acoustic models of automatic speech recognition (ASR) systems [SMA10] or for material classification [FHP15], among others. Covariance descriptors are also popular image or video descriptors [HHLS16].

Lastly, SPD matrices have found applications in deep learning. The authors of [GWB+19] show that an aggregation of learned deep convolutional features into an SPD matrix creates a robust representation of images which outperforms state-of-the-art methods for visual classification.

Manifold of SPD matrices

Let us recall the mathematical definition of the manifold of SPD matrices. The manifold of SPD matrices in n dimensions is embedded in the General Linear group of invertible matrices and defined as:

$$SPD = \left\{ S \in \mathbb{R}_{n \times n} : S^T = S, \forall z \in \mathbb{R}^n, z \neq 0, z^T S z > 0 \right\}.$$

The class `SPDMatricesSpace` inherits from the class `EmbeddedManifold` and has an `embedding_manifold` attribute which stores an object of the class `GeneralLinear`. SPD matrices in 2 dimensions can be visualized as ellipses with principal axes given by the eigenvectors of the SPD matrix, and the length of each axis proportional to the square-root of the corresponding eigenvalue. This is implemented in the `visualization` module of `geomstats`. We generate a toy data-set and plot it in Figure 4 with the following code snippet.

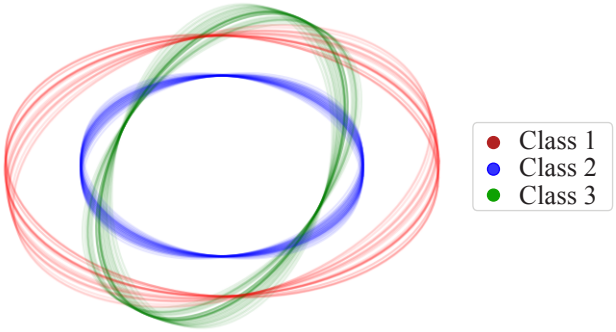


Fig. 4: *Simulated dataset of SPD matrices in 2 dimensions. We observe 3 classes of SPD matrices, illustrated with the colors red, green, and blue. The centroid of each class is represented by an ellipse of larger width.*

```
import geomstats.datasets.sample_sdp_2d as sampler

n_samples = 100
dataset_generator = sampler.DatasetSPD2D(
    n_samples, n_features=2, n_classes=3)

ellipsis = visualization.Ellipsis2D()
for i,x in enumerate(data):
    y = sampler.get_label_at_index(i, labels)
    ellipsis.draw(
        x, color=ellipsis.colors[y], alpha=.1)
```

Figure 4 shows a dataset of SPD matrices in 2 dimensions organized into 3 classes. This visualization helps in developing an intuition on the connectomes dataset that is used in the upcoming tutorial, where we will classify SPD matrices in 28 dimensions into 2 classes.

Classifying brain connectomes in Geomstats

We now delve into the tutorial in order to illustrate the use of traditional learning algorithms on the tangent spaces of manifolds implemented in `geomstats`. We use brain connectome data from the MSLP 2014 Schizophrenia Challenge. The connectomes are correlation matrices extracted from the time-series of resting-state fMRIs of 86 patients at 28 brain regions of interest: they are points on the manifold of SPD matrices in $n = 28$ dimensions. Our goal is to use the connectomes to classify patients into two classes: schizophrenic and control. First we load the connectomes and display two of them as heatmaps in Figure 5.

```
import geomstats.datasets.utils as data_utils

data, patient_ids, labels = \
    data_utils.load_connectomes()
```

Multiple metrics can be used to compute on the manifold of SPD matrices [DKZ09]. As mentionned in the previous tutorial, different metrics define different geodesics, exponential and logarithm maps and therefore different algorithms on a given manifold. Here, we import two of the most commonly used metrics on the SPD matrices, the log-Euclidean metric and the affine-invariant metric [PFA06], but we highlight that `geomstats` contains many more. We also check that our connectome data indeed belongs to the manifold of SPD matrices:

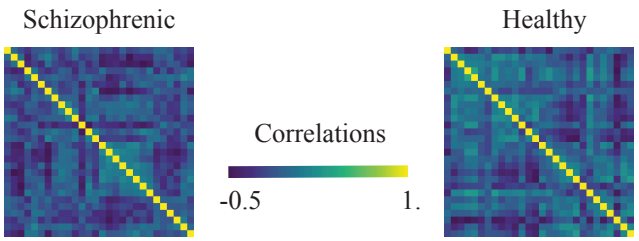


Fig. 5: *Subset of the connectomes dataset, available in* `geomstats` *with the function* `load_connectomes` *from the module* `datasets.utils`*. Connectomes are correlation matrices of 28 time-series extracted from fMRI data: they are elements of the manifold of SPD matrices in 28 dimensions. Left: connectome of a schizophrenic subject. Right: connectome of a healthy control.*

```
import geomstats.geometry.spd_matrices as spd

manifold = spd.SPDMatrices(n=28)
le_metric = spd.SPDMetricLogEuclidean(n=28)
ai_metric = spd.SPDMetricAffine(n=28)
logging.info(gs.all(manifold.belongs(data)))
```

```
INFO: True
```

Great! Now, although the sum of two SPD matrices is an SPD matrix, their difference or their linear combination with non-positive weights are not necessarily. Therefore we need to work in a tangent space of the SPD manifold to perform simple machine learning that relies on linear operations. The `preprocessing` module with its `ToTangentSpace` class allows to do exactly this.

```
from geomstats.learning.preprocessing import \
    ToTangentSpace
```

`ToTangentSpace` has a simple purpose: it computes the Fréchet Mean of the data set, and takes the logarithm map of each data point from the mean. This results in a data set of tangent vectors at the mean. In the case of the SPD manifold, these are simply symmetric matrices. `ToTangentSpace` then squeezes each symmetric matrix into a 1d-vector of size `dim = 28 * (28 + 1) / 2`, and outputs an array of shape `[n_connectomes, dim]`, which can be fed to your favorite `scikit-learn` algorithm.

We emphasize that `ToTangentSpace` computes the mean of the input data, and thus should be used in a pipeline (as e.g. `scikit-learn`'s `StandardScaler`) to avoid leaking information from the test set at train time.

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate

pipeline = make_pipeline(
    ToTangentSpace(le_metric), LogisticRegression(C=2))
```

We use a logistic regression on the tangent space at the Fréchet mean to classify connectomes, and evaluate the model with cross-validation. With the log-Euclidean metric we obtain:

```
result = cross_validate(pipeline, data, labels)
logging.info(result['test_score'].mean())
```

```
INFO: 0.67
```

And with the affine-invariant metric, replacing `le_metric` by `ai_metric` in the above snippet:

```
INFO: 0.71
```

We observe that the result depends on the metric. The Riemannian metric indeed defines the notion of the logarithm map, which is used to compute the Fréchet Mean and the tangent vectors corresponding to the input data points. Thus, changing the metric changes the result. Furthermore, some metrics may be more suitable than others for different applications. Indeed, we find published results that show how useful geometry can be with data on the SPD manifold (e.g [WAZF18], [NDV+14]).

We saw how to use the representation of points on the manifold as tangent vectors at a reference point to fit any machine learning algorithm, and we compared the effect of different metrics on the manifold of SPD matrices. Another class of machine learning algorithms can be used very easily on manifolds with `geomstats`: those relying on dissimilarity matrices. We can compute the matrix of pairwise Riemannian distances, using the *dist* method of the Riemannian metric object. In the following code-snippet, we use `ai_metric.dist` and pass the corresponding matrix `pairwise_dist` of pairwise distances to `scikit-learn`'s *K*-Nearest-Neighbors (KNN) classification algorithm:

```
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(
    metric='precomputed')

result = cross_validate(
    classifier, pairwise_dist, labels)
logging.info(result['test_score'].mean())
```

```
INFO: 0.72
```

This tutorial showed how to leverage `geomstats` to use standard learning algorithms for data on a manifold. In the next tutorial, we see a more complicated situation: the data points are not provided by default as elements of a manifold. We will need to use the low-level `geomstats` operations to design a method that embeds the dataset in the manifold of interest. Only then, we can use a learning algorithm.

Tutorial: Learning Graph Representations with Hyperbolic Spaces

Tutorial context and description

This tutorial demonstrates how to make use of the low-level geometric operations in `geomstats` to implement a method that embeds graph data into the hyperbolic space. Thanks to the discovery of hyperbolic embeddings, learning on Graph-Structured Data (GSD) has seen major achievements in recent years. It had been speculated for years that hyperbolic spaces may better represent GSD than Euclidean spaces [Gro87] [KPK+10] [BPK10] [ASM13]. These speculations have recently been shown effective through concrete studies and applications [NK17] [CCD17] [SDSGR18] [GZH+19]. As outlined by [NK17], Euclidean embeddings require large dimensions to capture certain complex relations such as the Wordnet noun hierarchy. On the other hand, this complexity can be captured by a lower-dimensional model of hyperbolic geometry such as the hyperbolic space of two dimensions [SDSGR18], also called the hyperbolic plane. Additionally, hyperbolic embeddings provide better visualizations of clusters on graphs than their Euclidean counterparts [CCD17].

This tutorial illustrates how to learn hyperbolic embeddings in `geomstats`. Specifically, we will embed the Karate Club graph dataset, representing the social interactions of the members of a university Karate club, into the Poincaré ball. Note that we will omit implementation details but an unabridged example and detailed notebook can be found on GitHub in the `examples` and `notebooks` directories of `geomstats`.

Hyperbolic spaces and machine learning applications

Before going into this tutorial, we review a few applications of hyperbolic spaces in the machine learning literature. First, Hyperbolic spaces arise in information and learning theory. Indeed, the space of univariate Gaussians endowed with the Fisher metric densities is a hyperbolic space [CSS05]. This characterization is used in various fields, for example in image processing, where each image pixel can be represented by a Gaussian distribution [AVF14], or in radar signal processing where the corresponding echo is represented by a stationary Gaussian process [ABY13]. Hyperbolic spaces can also be seen as continuous versions of trees and are therefore interesting when learning representations of hierarchical data [NK17]. Hyperbolic Geometric Graphs (HGG) have also been suggested as a promising model for social networks - where the hyperbolicity appears through a competition between similarity and popularity of an individual [PKS+12] and in learning communities on large graphs [GZH+19].

Hyperbolic space

Let us recall the mathematical definition of the hyperbolic space. The n-dimensional hyperbolic space H_n is defined by its embedding in the $(n+1)$-dimensional Minkowski space as:

$$H_n = \left\{x \in \mathbb{R}^{n+1} : -x_1^2 + ... + x_{n+1}^2 = -1\right\}. \quad (1)$$

In `geomstats`, the hyperbolic space is implemented in the class `Hyperboloid` and `PoincareBall`, which use different coordinate systems to represent points. These classes inherit from the class `EmbeddedManifold` and have an `embedding_manifold` attribute which stores an object of the class `Minkowski`. The 2-dimensional hyperbolic space is called the hyperbolic plane or Poincaré disk.

Learning graph representations with hyperbolic spaces in `geomstats`

Parameters and Initialization: We now proceed with the tutorial embedding the Karate club graph in a hyperbolic space. In the Karate club graph, each node represents a member of the club, and each edge represents an undirected relation between two members. We first load the Karate club dataset, display it in Figure 6 and print information regarding its nodes and vertices to provide insights into the graph's complexity.

```
karate_graph = data_utils.load_karate_graph()
nb_vertices_by_edges = (
    [len(e_2) for _, e_2 in
        karate_graph.edges.items()])
logging.info(
    'Number of vertices: %s', len(karate_graph.edges))
logging.info(
    'Mean edge-vertex ratio: %s',
    (sum(nb_vertices_by_edges, 0) /
        len(karate_graph.edges)))
```

```
INFO: Number of vertices: 34
INFO: Mean edge-vertex ratio: 4.588235294117647
```

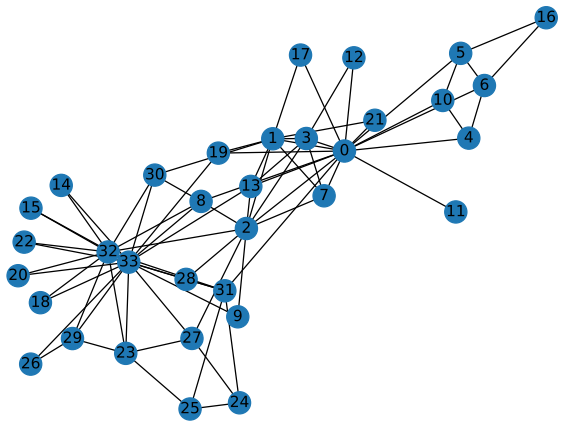


Fig. 6: *Karate club dataset, available in `geomstats` with the function `load_karate_graph` from the module `datasets.utils`. This dataset is a graph, where each node represents a member of the club and each edge represents a tie between two members of the club.*

Parameter	Description	Value
dim	Dimension of the hyperbolic space	2
max_epochs	Number of embedding iterations	15
lr	Learning rate	0.05
n_negative	Number of negative samples	2
context_size	Size of the context for each node	1
karate_graph	Instance of the `Graph` class returned by the function `load_karate_graph` in `datasets.utils`	

TABLE 1: *Hyperparameters used to embed the Karate Club Graph into a hyperbolic space.*

Table 1 defines the parameters needed to embed this graph into a hyperbolic space. The number of hyperbolic dimensions should be high ($n > 10$) only for graph datasets with a large number of nodes and edges. In this tutorial we consider a dataset with only 34 nodes, which are the 34 members of the Karate club. The Poincaré ball of two dimensions is therefore sufficient to capture the complexity of the graph. We instantiate an object of the class `PoincareBall` in `geomstats`.

```
from geomstats.geometry.poincare_ball
    import PoincareBall

hyperbolic_manifold = PoincareBall(dim=2)
```

Other parameters such as `max_epochs` and `lr` will be tuned specifically for each dataset, either manually leveraging visualization functions or through a grid/random search that looks for parameter values maximizing some performance function (a measure for cluster separability, normalized mutual information (NMI), or others). Similarly, the number of negative samples and context size are hyperparameters and will be further discussed below.

Learning the embedding by optimizing a loss function: Denote V as the set of nodes and $E \subset V \times V$ the set of edges of the graph. The goal of hyperbolic embedding is to provide a faithful and exploitable representation of the graph. This goal is mainly achieved by preserving first-order proximity that encourages nodes sharing edges to be close to each other. We can additionally pre-

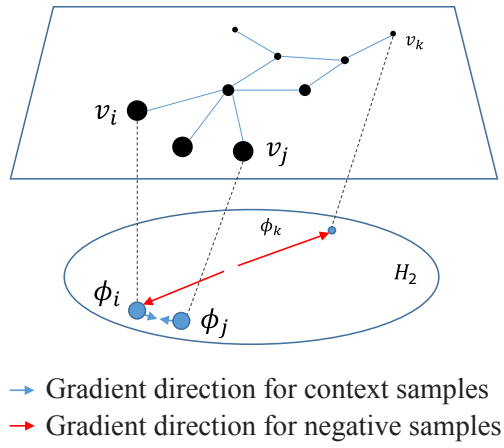


Fig. 7: *Embedding of the graph's nodes $\{v_i\}_i$ as points $\{\phi_i\}_i$ of the hyperbolic plane H_2, also called the Poincaré ball of 2 dimensions. The blue and red arrows represent the direction of the gradient of the loss function $\mathscr{L}$ from Equation 2. This brings context samples closer and separates negative samples.*

serve second-order proximity by encouraging two nodes sharing the "same context", i.e. not necessarily directly connected but sharing a neighbor, to be close. We define a context size (here equal to 1) and call two nodes "context samples" if they share a neighbor, and "negative samples" otherwise. To preserve first and second-order proximities, we adopt the following loss function similar to [NK17] and consider the "negative sampling" approach from [MSC+13]:

$$\mathscr{L} = -\sum_{v_i \in V} \sum_{v_j \in C_i} \left[\log(\sigma(-d^2(\phi_i, \phi'_j))) + \sum_{v_k \sim \mathscr{P}_n} \log(\sigma(d^2(\phi_i, \phi'_k))) \right] \tag{2}$$

where $\sigma(x) = (1+e^{-x})^{-1}$ is the sigmoid function and $\phi_i \in H_2$ is the embedding of the i-th node of V, C_i the nodes in the context of the i-th node, $\phi'_j \in H_2$ the embedding of $v_j \in C_i$. Negatively sampled nodes v_k are chosen according to the distribution $\mathscr{P}_n$ such that $\mathscr{P}_n(v) = (\deg(v)^{3/4}).(\sum_{v_i \in V} \deg(v_i)^{3/4})^{-1}$.

Intuitively one can see in Figure 7 that minimizing $\mathscr{L}$ makes the distance between ϕ_i and ϕ_j smaller, and the distance between ϕ_i and ϕ_k larger. Therefore by minimizing $\mathscr{L}$, one obtains representative embeddings.

Riemannian optimization: Following the literature on optimization on manifolds [GBH18], we use the following gradient updates to optimize $\mathscr{L}$:

$$\phi^{t+1} = \mathrm{Exp}_{\phi^t}\left(-lr \frac{\partial \mathscr{L}}{\partial \phi}\right)$$

where ϕ is a parameter of $\mathscr{L}$, $t \in \{1, 2, \cdots\}$ is the iteration number, and lr is the learning rate. The formula consists of first computing the usual gradient of the loss function for the direction in which the parameter should move. The Riemannian exponential map Exp is the operation introduced in the second tutorial: it takes a base point ϕ^t and a tangent vector T and returns the point ϕ^{t+1}. The Riemannian exponential map is a method of the `PoincareBallMetric` class in the `geometry` module of `geomstats`. It allows us to implement a straightforward generalization of standard gradient update in the Euclidean case. To compute the gradient of $\mathscr{L}$, we need to compute the gradients of: (i) the squared distance $d^2(x,y)$ on the hyperbolic space, (ii) the log sigmoid $\log(\sigma(x))$, and (iii) the composition of (i) with (ii).

For (i), we use the formula proposed by [ABY13] which uses the Riemannian logarithmic map. Like the exponential Exp, the logarithmic map is implemented under the `PoincareBallMetric`.

```
def grad_squared_distance(point_a, point_b, manifold):
    log = manifold.metric.log(point_b, point_a)
    return -2 * log
```

For (ii), we compute the well-known gradient of the logarithm of the sigmoid function as: $(\log\sigma)'(x) = (1+\exp(x))^{-1}$. For (iii), we apply the composition rule to obtain the gradient of $\mathscr{L}$. The following function computes $\mathscr{L}$ and its gradient on the context samples, while ignoring the part dealing with the negative samples for simplicity of exposition. The code implementing the whole `loss` function is available on GitHub.

```
def loss(example, context_embedding, manifold):

    context_distance = manifold.metric.squared_dist(
        example, context_embedding)
    context_loss = log_sigmoid(-context_distance)
    context_log_sigmoid_grad = -grad_log_sigmoid(
        -context_distance)

    context_distance_grad = grad_squared_distance(
        example, context_embedding, manifold)

    context_grad = (context_log_sigmoid_grad
        * context_distance_grad)

    return context_loss, -context_grad
```

Capturing the graph structure: We perform initialization computations that capture the graph structure. We compute random walks initialized from each v_i up to some length (five by default). The context nodes v_j will be later picked from the random walk of v_i.

```
random_walks = karate_graph.random_walk()
```

Negatively sampled nodes v_k are chosen according to the previously defined probability distribution function $\mathscr{P}_n(v_k)$ implemented as

```
negative_table_parameter = 5
negative_sampling_table = []

for i, nb_v in enumerate(nb_vertices_by_edges):
    negative_sampling_table += (
        [i] * int((nb_v**(3. / 4.)))
            * negative_table_parameter)
```

Numerically optimizing the loss function: We can now embed the Karate club graph into the Poincaré disk. The details of the initialization are provided on GitHub. The array `embeddings` contains the embeddings ϕ_i's of the nodes `v_i`'s of the current iteration. At each iteration, we compute the gradient of $\mathscr{L}$. The graph nodes are then moved in the direction pointed by the gradient. The movement of the nodes is performed by following geodesics in the Poincaré disk in the gradient direction. In practice, the key to obtaining a representative embedding is to carefully tune the learning rate so that all of the nodes make small movements at each iteration.

A first level loop iterates over the epochs while the table `total_loss` records the value of $\mathscr{L}$ at each iteration. A second

level nested loop iterates over each path in the previously computed random walks. Observing these walks, note that nodes having many edges appear more often. Such nodes can be considered as important crossroads and will therefore be subject to a greater number of embedding updates. This is one of the main reasons why random walks have proven to be effective in capturing the structure of graphs. The context of each v_i will be the set of nodes v_j belonging to the random walk from v_i. The `context_size` specified earlier will limit the length of the walk to be considered. Similarly, we use the same `context_size` to limit the number of negative samples. We find ϕ_i from the `embeddings` array.

A third and fourth level nested loops will iterate on each v_j and v_k. From within, we find ϕ'_j and ϕ'_k and call the `loss` function to compute the gradient. Then the Riemannian exponential map is applied to find the new value of ϕ_i as we mentioned before.

```
for epoch in range(max_epochs):
    total_loss = []
    for path in random_walks:
        for example_index,
                one_path in enumerate(path):
            context_index = path[max(
                0, example_index - context_size):
                min(example_index + context_size,
                len(path))]
            negative_index = gs.random.randint(
                negative_sampling_table.shape[0],
                size=(len(context_index), n_negative))
            negative_index = (
                negative_sampling_table[negative_index])
            example_embedding = embeddings[one_path]
            for one_context_i, one_negative_i in \
                zip(context_index, negative_index):
                context_embedding = (
                    embeddings[one_context_i])
                negative_embedding = (
                    embeddings[one_negative_i])
                l, g_ex = loss(
                    example_embedding,
                    context_embedding,
                    negative_embedding,
                    hyperbolic_manifold)
                total_loss.append(l)

                example_to_update = (
                    embeddings[one_path])
                embeddings[one_path] = (
                    hyperbolic_metric.exp(
                    -lr * g_ex, example_to_update))
    logging.info(
        'iteration %d loss_value %f',
        epoch, sum(total_loss, 0) / len(total_loss))
```

```
INFO: iteration 0 loss_value 1.819844
INFO: iteration 14 loss_value 1.363593
```

Figure 8 shows the graph embedding at different iterations with the true labels of each node represented with color. Notice how the embedding at convergence separates well the two clusters. Thus, it seems that we have found a useful representation of the graph.

To demonstrate the usefulness of the embedding learned, we show how to apply a K-means algorithm in the hyperbolic plane to predict the label of each node in an unsupervised approach. We use the `learning` module of `geomstats` and instantiate an object of the class `RiemannianKMeans`. Observe again how `geomstats` classes follow `scikit-learn`'s API. We set the number of clusters and plot the results.

```
from geomstats.learning.kmeans import RiemannianKMeans

kmeans = RiemannianKMeans(
```

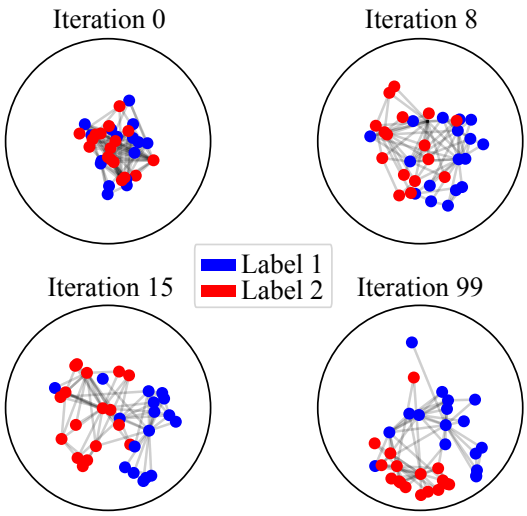


Fig. 8: Embedding of the Karate club graph into the hyperbolic plane at different iterations. The colors represent the true label of each node.

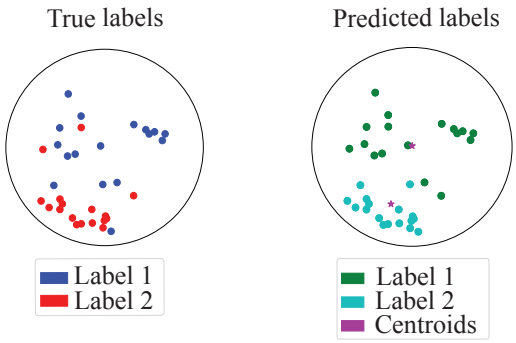


Fig. 9: Results of the Riemannian K-means algorithm on the Karate graph dataset embedded in the hyperbolic plane. Left: True labels associated to the club members. Right: Predicted labels via Riemannian K-means on the hyperbolic plane. The centroids of the clusters are shown with a star marker.

```
    hyperbolic_manifold.metric, n_clusters=2,
    mean_method='frechet-poincare-ball')
centroids = kmeans.fit(X=embeddings, max_iter=100)
labels = kmeans.predict(X=embeddings)
```

Figure 9 shows the true labels versus the predicted ones: the two groups of the karate club members have been well separated!

Conclusion

This paper demonstrates the use of `geomstats` in performing geometric learning on data belonging to manifolds. These tutorials, as well as many other learning examples on a variety of manifolds, can be found at `geomstats.ai`. We hope that this hands-on presentation of Geometric Learning will help to further democratize the use of differential geometry in the machine learning community.

Acknowledgements

This work is partially supported by the National Science Foundation, grant NSF DMS RTG 1501767, the Inria-Stanford associated team GeomStats, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement G-Statistics No. 786854).

REFERENCES

[ABY13] Marc Arnaudon, Frédéric Barbaresco, and Le Yang. Riemannian medians and means with applications to radar signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 7(4):595–604, 2013. URL: https://ieeexplore.ieee.org/document/6514112, doi:10.1109/JSTSP.2013.2261798.

[AS14] Dena Asta and Cosma Rohilla Shalizi. Geometric Network Comparison. *Journal of Machine Learning Research*, 2014. URL: http://arxiv.org/abs/1411.1350, doi:10.1109/PES.2006.1709566.

[ASM13] Aaron B Adcock, Blair D Sullivan, and Michael W Mahoney. Tree-like structure in large social and information networks. In *2013 IEEE 13th International Conference on Data Mining*, pages 1–10. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6729484, doi:10.1109/ICDM.2013.77.

[AVF14] Jesus Angulo and Santiago Velasco-Forero. Morphological processing of univariate Gaussian distribution-valued images based on Poincaré upper-half plane representation. In Frank Nielsen, editor, *Geometric Theory of Information*, Signals and Communication Technology, pages 331–366. Springer International Publishing, 5 2014. URL: https://hal.archives-ouvertes.fr/hal-00795012, doi:10.1007/978-3-319-05317-2_12.

[Bar15] Alexandre Barachant. PyRiemann: Python package for covariance matrices manipulation and Biosignal classification with application in Brain Computer interface, 2015. URL: https://github.com/alexandrebarachant/pyRiemann, doi:10.5281/zenodo.3715511.

[BBL+17] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. URL: https://ieeexplore.ieee.org/document/7974879, doi:10.1109/MSP.2017.2693418.

[BG18] Gary Bécigneul and Octavian-Eugen Ganea. Riemannian Adaptive Optimization Methods. In *Proc. of ICLR 2019*, pages 1–16, 2018. URL: http://arxiv.org/abs/1810.00760.

[BPK10] Marián Boguná, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature communications*, 1(1):1–8, Oct 2010. URL: https://www.nature.com/articles/ncomms1063, doi:10.1038/ncomms1063.

[CBA15] Emmanuel Chevallier, Frédéric Barbaresco, and Jesus Angulo. Probability density estimation on the hyperbolic space applied to radar processing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9389:753–761, 2015. URL: https://link.springer.com/chapter/10.1007/978-3-319-25040-3_80, doi:10.1007/978-3-319-25040-3_80.

[CCD17] Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *13th International Workshop on Mining and Learning with Graphs*, 2017. URL: https://arxiv.org/abs/1705.10359.

[Cen12] Andrea Censi. PyGeometry: Library for handling various differentiable manifolds., 2012. URL: https://github.com/AndreaCensi/geometry.

[CS16] Anoop Cherian and Suvrit Sra. Positive Definite Matrices: Data Representation and Applications to Computer Vision. In *Algorithmic Advances in Riemannian Geometry and Applications*. Springer, 2016. URL: https://www.springerprofessional.de/en/positive-definite-matrices-symmetric-positive-definite-spd-matri/10816206, doi:10.1007/978-3-319-45026-1.

[CSS05] Sueli IR Costa, Sandra A Santos, and João E Strapasson. Fisher information matrix and hyperbolic geometry. In *IEEE Information Theory Workshop, 2005.*, pages 3–pp. IEEE, 2005. URL: https://ieeexplore.ieee.org/document/1531851, doi:10.1109/ITW.2005.1531851.

[DKZ09] Ian L. Dryden, Alexey Koloydenko, and Diwei Zhou. Non-Euclidean statistics for covariance matrices, with applications to diffusion tensor imaging. *Annals of Applied Statistics*, 3(3):1102–1123, September 2009. Publisher: Institute of Mathematical Statistics. URL: https://projecteuclid.org/euclid.aoas/1254773280, doi:10.1214/09-AOAS249.

[FHP15] Masoud Faraki, Mehrtash T Harandi, and Fatih Porikli. Material Classification on Symmetric Positive Definite Manifolds. In *2015 IEEE Winter Conference on Applications of Computer Vision*, pages 749–756, 1 2015. URL: https://ieeexplore.ieee.org/document/7045959, doi:10.1109/WACV.2015.105.

[GBH18] Octavian Ganea, Gary Becigneul, and Thomas Hofmann. Hyperbolic neural networks. In *Advances in Neural Information Processing Systems 31 (NIPS)*, pages 5345–5355. Curran Associates, Inc., 2018. URL: http://papers.nips.cc/paper/7780-hyperbolic-neural-networks.pdf.

[Gro87] Mikhail Gromov. *Hyperbolic Groups*, pages 75–263. Springer New York, New York, NY, 1987. URL: https://link.springer.com/chapter/10.1007/978-1-4613-9586-7_3, doi:10.1007/978-1-4613-9586-7_3.

[GWB+19] Zhi Gao, Yuwei Wu, Xingyuan Bu, Tan Yu, Junsong Yuan, and Yunde Jia. Learning a robust representation via a deep network on symmetric positive definite manifolds. *Pattern Recognition*, 92:1–12, August 2019. URL: https://linkinghub.elsevier.com/retrieve/pii/S0031320319301062, doi:10.1016/j.patcog.2019.03.007.

[GZH+19] Thomas Gerald, Hadi Zaatiti, Hatem Hajri, Nicolas Baskiotis, and Olivier Schwander. From node embedding to community embedding : A hyperbolic approach, 2019. URL: https://arxiv.org/abs/1907.01662, arXiv:1907.01662.

[HHLS16] M. T. Harandi, R. Hartley, B. Lovell, and C. Sanderson. Sparse coding on symmetric positive definite manifolds using bregman divergences. *IEEE Transactions on Neural Networks and Learning Systems*, 27(6):1294–1306, 2016. doi:10.1109/TNNLS.2014.2387383.

[HKKM10] Stephan Huckemann, Peter Kim, Ja Yong Koo, and Axel Munk. Möbius deconvolution on the hyperbolic plane with application to impedance density estimation. *Annals of Statistics*, 38(4):2465–2498, 2010. URL: https://projecteuclid.org/euclid.aos/1278861254, doi:10.1214/09-AOS783.

[HVS+16] Junpyo Hong, Jared Vicory, Jörn Schulz, Martin Styner, J S Marron, and StephenM Pizer. Non-Euclidean Classification of Medically Imaged Objects via s-reps. *Med Image Anal*, 31:37–45, 2016. URL: https://www.sciencedirect.com/science/article/abs/pii/S1361841516000141, doi:10.1016/j.media.2016.01.007.

[JDM12] Sungkyu Jung, Ian L. Dryden, and J. S. Marron. Analysis of principal nested spheres. *Biometrika*, 99(3):551–568, 2012. URL: http://www.statistics.pitt.edu/sungkyu/papers/Biometrika-2012-Jung-551-68.pdf, doi:10.1093/biomet/ass022.

[KH05] John T Kent and Thomas Hamelryck. Using the Fisher-Bingham distribution in stochastic models for protein structure. *Quantitative Biology, Shape Analysis, and Wavelets*, 24(1):57–60, 2005. URL: http://www.amsta.leeds.ac.uk/statistics/workshop/lasr2005/Proceedings/kent.pdf.

[Koc19] Maxim Kochurov. Geoopt: Riemannian Adaptive Optimization Methods with pytorch optim, 2019. URL: https://arxiv.org/abs/2005.02819.

[KPK+10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82:036106, Sep 2010. URL: https://journals.aps.org/pre/abstract/10.1103/PhysRevE.82.036106, doi:10.1103/PhysRevE.82.036106.

[KS17] Line Kühnel and Stefan Sommer. Computational Anatomy in Theano. *CoRR*, 2017. URL: https://link.springer.com/chapter/10.1007/978-3-319-67675-3_15, doi:10.1007/978-3-319-67675-3_15.

[MGLB+] Nina Miolane, Nicolas Guigui, Alice Le Brigant, Johan Mathe, Benjamin Hou, Yann Thanwerdas, Stefan Heyder, Olivier Peltre, Nicolas Koep, Hadi Zaatiti, Hatem Hajri, Yann Cabanes, Thomas Gerald, Paul Chauchat, Daniel Brooks, Christian Shewmake, Bernhard Kainz, Claire Donnat, Susan Holmes, and Xavier Pennec. Geomstats : a Python Package for Riemannian Geometry in Machine Learning. URL: https://arxiv.org/abs/2004.04667.

[MJK+18] Mayank Meghwanshi, Pratik Jawanpuria, Anoop Kunchukuttan, Hiroyuki Kasai, and Bamdev Mishra. McTorch, a manifold optimization library for deep learning, 2018. URL: http://arxiv.org/abs/1810.01811.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26 (NIPS)*, pages 3111–3119. Curran Associates, Inc., 2013. URL: https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality, doi:https://dl.acm.org/doi/10.5555/2999792.2999959.

[NDV+14] Bernard Ng, Martin Dressler, Gaël Varoquaux, Jean Baptiste Poline, Michael Greicius, and Bertrand Thirion. Transport on Riemannian manifold for functional connectivity-based classification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8674 LNCS, pages 405–412, Cham, 2014. Springer International Publishing. URL: http://link.springer.com/10.1007/978-3-319-10470-6{_}51, doi:10.1007/978-3-319-10470-6_51.

[NK17] Maximillian Nickel and Douwe Kiela. Poincaré Embeddings for Learning Hierarchical Representations. In I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6338–6347. Curran Associates, Inc., 2017. URL: http://papers.nips.cc/paper/7213-poincare-embeddings-for-learning-hierarchical-representations.pdf.

[PFA06] Xavier Pennec, Pierre Fillard, and Nicholas Ayache. A Riemannian Framework for Tensor Computing. *International Journal of Computer Vision*, 66(1):41–66, 1 2006. URL: https://link.springer.com/article/10.1007/s11263-005-3222-z, doi:10.1007/s11263-005-3222-z.

[PKS+12] Fragkiskos Papadopoulos, Maksim Kitsak, M Ángeles Serrano, Marián Boguná, and Dmitri Krioukov. Popularity versus similarity in growing networks. *Nature*, 489(7417):537–540, 2012. URL: https://www.nature.com/articles/nature11459, doi:10.1038/nature11459.

[Pos01] Mikhail Postnikov. *Riemannian Geometry*. Encyclopaedia of Mathem. Sciences. Springer, 2001. URL: https://encyclopediaofmath.org/wiki/Riemannian_geometry, doi:10.1007/978-3-662-04433-9.

[PSF19] Xavier Pennec, Stefan Sommer, and Tom Fletcher. *Riemannian Geometric Statistics in Medical Image Analysis*. Elsevier Ltd, first edit edition, 2019. URL: https://www.elsevier.com/books/riemannian-geometric-statistics-in-medical-image-analysis/pennec/978-0-12-814725-2, doi:10.1016/C2017-0-01561-6.

[SDSGR18] Frederic Sala, Chris De Sa, Albert Gu, and Christopher Re. Representation tradeoffs for hyperbolic embeddings. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4460–4469, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL: http://proceedings.mlr.press/v80/sala18a.html.

[SMA10] Yusuke Shinohara, Takashi Masuko, and Masami Akamine. Covariance clustering on Riemannian manifolds for acoustic model compression. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4326–4329, 3 2010. URL: https://ieeexplore.ieee.org/document/5495661, doi:10.1109/ICASSP.2010.5495661.

[STK05] Olaf Sporns, Giulio Tononi, and Rolf Kötter. The human connectome: A structural description of the human brain. *PLOS Computational Biology*, 1(4):0245–0251, 09 2005. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1239902/, doi:10.1371/journal.pcbi.0010042.

[TKW16] James Townsend, Niklas Koep, and Sebastian Weichwald. Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. *Journal of Machine Learning Research*, 17(137):1–5, 2016. URL: http://jmlr.org/papers/v17/16-177.html, doi:https://dl.acm.org/doi/10.5555/2946645.3007090.

[WAZF18] Eleanor Wong, Jeffrey S. Anderson, Brandon A. Zielinski, and P. Thomas Fletcher. Riemannian Regression and Classification Models of Brain Networks Applied to Autism. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11083 LNCS of *Lecture Notes in Computer Science*, pages 78–87. Springer International Publishing, 2018. URL: https://link.springer.com/chapter/10.1007/978-3-030-00755-3_9, doi:10.1007/978-3-030-00755-3_9.

[Wyn14] Kieran Wynn. PyQuaternions: A fully featured, pythonic library for representing and using quaternions, 2014. URL: https://github.com/KieranWynn/pyquaternion.

[WZD+13] Jinhui Wang, Xinian Zuo, Zhengjia Dai, Mingrui Xia, Zhilian Zhao, Xiaoling Zhao, Jianping Jia, Ying Han, and Yong He. Disrupted functional brain connectome in individuals at risk for Alzheimer's disease. *Biological Psychiatry*, 73(5):472–481, 2013. URL: http://dx.doi.org/10.1016/j.biopsych.2012.03.026, doi:10.1016/j.biopsych.2012.03.026.

[YZLM12] Ying Yuan, Hongtu Zhu, Weili Lin, and J S Marron. Local polynomial regression for symmetric positive definite matrices. *Journal of the Royal Statistical Society Series B*, 74(4):697–719, 2012. URL: https://econpapers.repec.org/RePEc:bla:jorssb:v:74:y:2012:i:4:p:697-719, doi:10.1111/j.1467-9868.2011.01022.x.

Network visualizations with Pyvis and VisJS

Giancarlo Perrone‡*, Jose Unpingco‡, Haw-minn Lu‡

Abstract—Pyvis is a Python module that enables visualizing and interactively manipulating network graphs in the Jupyter notebook, or as a standalone web application. Pyvis is built on top of the powerful and mature VisJS JavaScript library, which allows for fast and responsive interactions while also abstracting away the low-level JavaScript and HTML. This means that elements of the rendered graph visualization, such as node/edge attributes can be specified within Python and shipped to the JavaScript layer for VisJS to render. This declarative approach makes it easy to quickly explore graph visualizations and investigate data relationships. In addition, Pyvis is highly customizable so that colors, sizes, and hover tooltips can be assigned to the rendered graph. The network graph layout is controlled by a front-end physics engine that is configurable from a Python interface, allowing for the detailed placement of the graph elements. In this paper, we outline use cases for Pyvis with specific examples to highlight key features for any analysis workflow. A brief overview of Pyvis' implementation describes how the Python front-end binding uses simple Pyvis calls.

Index Terms—networks, graphs, relationship

Introduction

Successful Data Science is about discovering meaningful relationships in data. Visually representing these relationships using a network graph helps to accelerate understanding and make data driven decisions. Many research areas take advantage of the insight that network analysis techniques can offer. Fields in social networking, cognitive studies, telecommunications, and biological systems all leverage the applications of network theory and computation. Representing these relationships using a network graph is fundamental to all approaches, but generating an interactive and fluid graph visualization can be challenging, especially for large datasets. We introduce Pyvis, based upon the mature VisJS [vis20b] JavaScript library which enables fluid and interactive visualizations of complex network graphs. Pyvis seeks to simplify the interactive process by implementing an existing JavaScript graphics library to abstract away the low-level front end components, leaving the construction of these network data structures to Python.

The Pyvis network data structure matches the JavaScript VisJS object. This makes it easy to interpret and implement the underlying data structures from the Python layer, since the actual front end component is generated by the JavaScript library. A resulting static HTML document shows the network graph, with interactions such as dragging, zooming, hovering, and clicking. These interactions help visualize dense complex networks that are hard to explore using static graphics.

Before open-sourcing Pyvis, we used it successfully to understand relationships among hundreds of variables in a complex survey. Although we maintained an efficient data structure to represent the trends in the survey responses, we still needed a way to visualize and interact with additional metadata. Pyvis made it easy to abstract our existing data structure into nodes and edges with our desired metadata and then render the visualization with VisJS to easily identify the interrelationships. In this paper, we describe the design of Pyvis with examples showing the data structures which are rendered by VisJS.

In the following section, we demonstrate how to get up and running with Pyvis in a smaller scope by showing off the common methods of creating a network. This will also include some exposure to the customizability options that makes Pyvis so useful.

In the Layout section, we will see exactly how nodes and edges can be spatially specified by interacting with various physics parameters interpreted by the front end engine.

Integrations with Jupyter and NetworkX will be presented to establish Pyvis compatibility with popular data science workflows.

Finally, a thought out example will include the interpretation of a practical Game of Thrones relationship dataset to demonstrate a Pyvis use case from the ground up. This minimal example will be a base case for the features that Pyvis supports.

Pyvis Usage

Installing Pyvis is straight-forward with details at the project documentation website [Gia18]. All of the following examples will utilize familiar Python data structures with some connections to the popular and powerful NetworkX package [HSS08]. The basic `Network` class is the container for graph and front end properties. All networks must be instantiated as a `Network` class instance:

```
from pyvis.network import Network
g = Network()
```

Nodes can be added by providing an integer or string `id` and an optional label.

```
g.add_node(1)
g.add_node(2)
print(g)
```

```
{
 "Nodes": [
     1,
     2
 ],
 "Edges": [],
 "Height": "500px",
```

* Corresponding author: gperrone@westhealth.org
‡ Gary and Mary West Health Institute

Fig. 1: *Multiple nodes and attributes added at once*

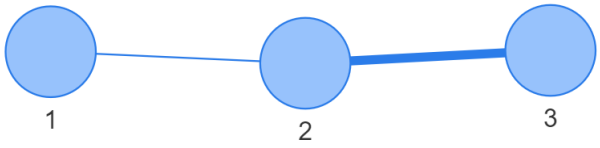


Fig. 2: *Edges with a custom weight*

```
 "Width": "500px"
}
```

The `add_nodes` method consumes a list of nodes (**Fig** 1):

```
nodes = ["a", "b", "c", "d"]
g.add_nodes(nodes)
g.add_nodes("hello")
```

Keyword arguments can be used to add properties to the nodes in `Network`:

```
g = Network()
g.add_nodes(
   [1,2,3],
   value=[10, 100, 400], # values adjust node size
   x=[21.4, 154.2, 11.2],
   y=[100.2, 23.54, 32.1],
   label=["NODE 1", "NODE 2", "NODE 3"],
   color=["#00ff1e", "#162347", "#dd4b39"]
)
g.show("example.html")
```

The following node properties influence the resulting visualization:

- size - The raw circumference of a single node
- value - Circumference of node but scaled according to all values
- title - The title displays over each node while mousing over it
- x - X coordinate of node for custom layouts
- y - Y coordinate of node for custom layouts
- label - A label appearing under each node
- color - The color of the node

Nodes must exist in the network instance in order to add edges

```
g.add_edge(1, 2)
# will adjust edge thickness
g.add_edge(2, 3, weight=5)
```

Edges can be added all at once by supplying a list of tuples to a call to *add_edges()*. The following is an equivalent result (**Fig** 2):

```
g.add_edges([(1, 2), (2, 3, 5)])
g.show("example.html")
```

Notice how an optional element is included in the 3-tuple above (2, 3, 5) representing the weight of the edge. This additional edge data allows for expressing weighted networks and is clearly noticeable in the visualization.

Layout

In situations where your network involves complex connections, Pyvis allows you to manually explore these relationships with intuitive mouse interactions. Nodes can be dragged into more visible positions if the view is obstructed.

All of this is made possible by the front end engine provided by VisJS. Their extensive documentation defines several options for supplying layout and physics configurations to instances of a network. These physics options are fundamental to VisJS, so tweaking the physics of the rendered simulation is as simple as providing the parameters to the specific solver.

The physics options dictates how a user can interact with the objects in the graph. The intent of the physic options is to make manipulating graph objects feel more intuitive when moving nodes around. As an example, the user can manipulate a portion of a graph that is densely populated to view a graph segment of the interest more clearly. VisJS implements several physical simulations such as Barnes Hut [BH86]. Others are mentioned in the VisJS documentation [vis20a].

We can configure the physics engine from within Pyvis:

```
g = Network()
# physics solvers supported:
# barnesHut, forceAtlas2Based, repulsion,
# hierarchicalRepulsion
g.barnes_hut(
 gravity=-80000,
 central_gravity=0.3,
 spring_length=250,
 spring_strength=0.001,
 damping=0.09,
 overlap=0,
)
print(g.options.physics)
{'enabled': True,
'stabilization':
<pyvis.physics.Physics.Stabilization
object at 0x7f99e6a03f90>,
'barnesHut': <pyvis.physics.Physics.barnesHut
object at 0x7f99e6de3710>}
```

In order to avoid the scenario of "guessing" parameter values for an optimal network physics configuration, VisJS offers a useful interaction for experimenting with theses values.

These interactions are enabled via Pyvis (**Fig** 3):

```
# choose to only show the physics options
g.show_buttons(filter_=["physics"])
```

Here, we choose to display the options for the physics component of the network. Omitting a filter in the call will display the configuration of the entire network including nodes, edges, layout, and interaction. The JSON options displayed in the visualization represent the current configuration depending on the displayed sliders. You can copy/paste those options to supply your network with custom settings:

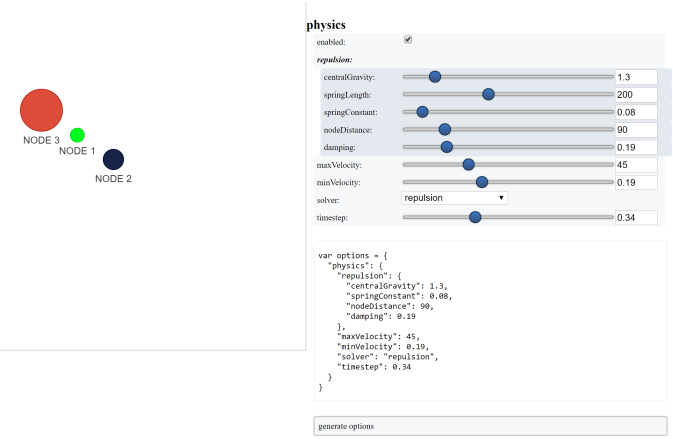


Fig. 3: *Live layout GUI with physics filter*

```
g.set_options(
    """
    var options = {
        "physics": {
            "repulsion": {
                "centralGravity": 1.3,
                "springConstant": 0.08,
                "nodeDistance": 90,
                "damping": 0.19
            },
            "maxVelocity": 45,
            "minVelocity": 0.19,
            "solver": "repulsion",
            "timestep": 0.34
        }
    }
    """
)
print(g.options)
```

```
{'physics': {'repulsion': {'centralGravity': 1.3,
'springConstant': 0.08,
'nodeDistance': 90,
'damping': 0.19},
'maxVelocity': 45,
'minVelocity': 0.19,
'solver': 'repulsion',
'timestep': 0.34}}
```

The methods of a `Network` instance construct an internal structure compatible with VisJS, demonstrated by the consistent pattern of JSON outputs seen above.

NetworkX Support

Although Pyvis supports its own methods for constructing a network data structure, you might feel more comfortable using the more established and dedicated NetworkX package. Pyvis allows you to define a NetworkX graph instance to then supply it to Pyvis (**Fig** 4).

```
import networkx as nx
from pyvis.network import Network

nxg = nx.random_tree(20)
g=Network(directed=True)
g.from_nx(nxg)
g.show("networkx.html")
```

Pyvis current behavior recognizes the basic topology of a NetworkX graph, not accounting for any custom attributes

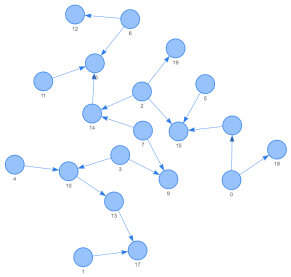


Fig. 4: *NetworkX graph rendered with Pyvis*

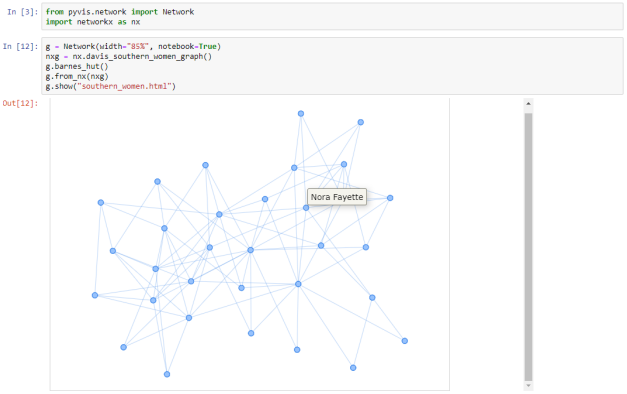


Fig. 5: *Network rendered in Jupyter Notebook cell*

provided. Any other attributes like node color, size, and layout would need to be manually added to the resulting Pyvis graph.

Future plans are to fully integrate NetworkX graphs to fully interpret them, preserving attributes in the resulting Pyvis visualizations.

Jupyter Support

For efficient prototyping of visualized graphs, Pyvis aims to utilize Jupyter's front-end IFrame features to embed the graph in a notebook output cell. With that in mind, embedding a Pyvis visualization into a Jupyter notebook is essentially the same as described above. The only difference is that one should pass in a notebook argument during instantiation. The result of the visualization is shown in the output cell below the `show()` invocation. Pyvis upon the call to `show()` writes the HTML that serves an IFrame, which displays the result in the output cell (**Fig** 5).

One thing to keep in mind is that an HTML file is always generated due to the dependence on the VisJS JavaScript bindings.

Example

To get a better understanding of the flow of a typical Pyvis network visualization, we can take a look at the following code snippet to show off a typical application of the features. I have taken a Game of Thrones dataset ([Bev] Storm of Swords Dataset) defining the relationships between characters and the frequencies between them to create a network to naturally

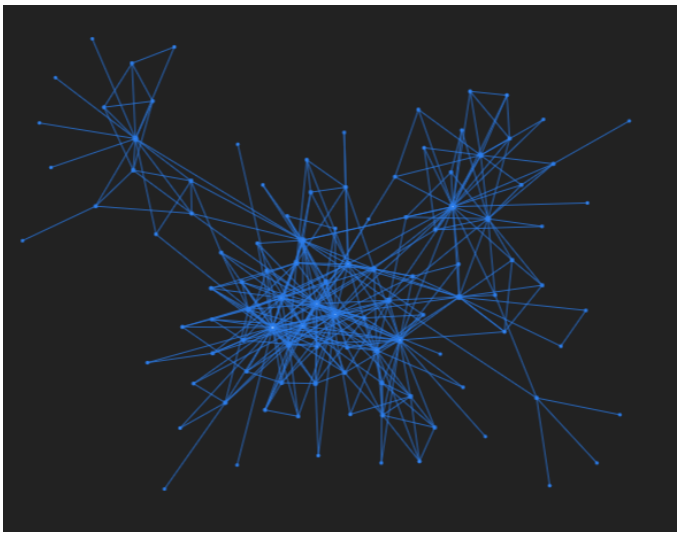


Fig. 6: *Game of Thrones network dictates relationships between characters*

express this. Specifically, it is a csv file containing pairs of characters and a weight between them. The final visualization contains 107 nodes. (**Fig** 6)

```
from pyvis.network import Network
import pandas as pd

got_net = Network(
   height="750px",
   width="100%",
   bgcolor="#222222",
   font_color="white"
)

# set the physics layout of the network
got_net.barnes_hut()
got_data = pd.read_csv("stormofswords.csv")

sources = got_data['Source']
targets = got_data['Target']
weights = got_data['Weight']

edge_data = zip(sources, targets, weights)

for e in edge_data:
   src = e[0]
   dst = e[1]
   w = e[2]

   got_net.add_node(src, src, title=src)
   got_net.add_node(dst, dst, title=dst)
   got_net.add_edge(src, dst, value=w)

neighbor_map = got_net.get_adj_list()

# add neighbor data to node hover data
for node in got_net.nodes:
   node["title"] += " Neighbors:<br>" + \
            "<br>".join(neighbor_map[node["id"]])
   node["value"] = len(neighbor_map[node["id"]])

got_net.show("gameofthrones.html")
```

In the network, the size of a node correlates to the number of relationships it contains. This calculation benefits from the use of an adjacency list to easily record the information pertaining to each node's neighbors. To see this, the character "Tyrion"

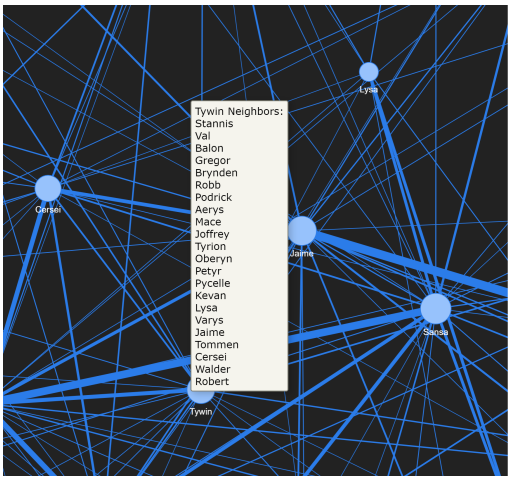


Fig. 7: *Zooming into Game of Thrones network offers concise view*

contains many connections, resulting in a larger node. Furthermore, Tyrion's strongest connections are easily noticed by the thick edges, and it is easy to see that Tyrion and Sansa are a strong relationship in the network.

At a glance, the resulting relationship network looks too intertwined to make any practical conclusions. However, the beauty of Pyvis is that each and every component of the network can be focused. For example, zooming in to a dense portion of the network, we can hover over a particular node to get a glimpse of the scenario. (**Fig** 7)

This hover tooltip offers the context behind a particular node. We can see the immediate neighbors for each and every node since we provided a *title* attribute during the network construction. This simple example can be expanded upon to create more custom interactions tailored to specific needs of a dataset.

The network also uses weights. By providing a *value* attribute to each node we can see these values being represented by a node's size. In the code I used the amount of neighbors to dictate the node weight. This is a strong visual cue which makes it easy to see which nodes have the most connections.

The edge weights are assigned in a similar manner, although the dataset already provided the connection strength between nodes. These edge weights are distinguishable in the final visualization, once again proving the usefulness of Pyvis' front-end features.

Under the Hood

VisJS reduces the definition of a network to a declarative set of objects. Nodes, Edges, and an Options JSON object are given to the VisJS Network constructor. The following basic example from their documentation proves this:

```
// create an array with nodes
var nodes = new vis.DataSet([
   {id: 1, label: 'Node 1'},
   {id: 2, label: 'Node 2'},
]);

// create an array with edges
var edges = new vis.DataSet([
   {from: 1, to: 2},
]);
```

```
// create a network
var container = document.getElementById('mynetwork');

// provide the data in the vis format
var data = {
   nodes: nodes,
   edges: edges
};
var options = {};

// initialize your network!
var network = new vis.Network(container, data, options);
```

This pattern makes Jinja [Pro] templating an obvious candidate for generalizing a set of JavaScript declarations. VisJS documentation provides a complete set of supported attributes for each data structure, so incorporating them into the Python layer involves representing each object as Python objects which are then serialized and sent to Jinja to handle the templating.

A simple example of this process in action is outlined below:

```
self.html = template.render(nodes=nodes, edges=edges)
```

In this case, a template HTML file is rendered with node and edge data matching a format compatible with a VisJS Network instance.

Conclusion

Pyvis is a powerful python module for visualizing and interactively manipulating network graphs in a standalone web application or a Jupyter notebook. Pyvis brings the power of VisJS to Python, thus enabling data scientists who use Jupyter to interactively visualize network graphs with all the fluid interactions of a pure-JavaScript application. Future directions for Pyvis include supporting the front end interactivity with more JavaScript enabled features, and optimization/caching of node positions for larger networks. Those with JavaScript and VisJS knowledge would be able to provide insight towards prospective front end features, since these will leverage actual VisJS references. As Pyvis use case grows in scope, additional features and suggestions will be requested, paving the path for a robust version of Pyvis meeting user experience expectations.

Code samples presented here, and with the corresponding poster presentation, as well as other supplemental material are available at West Health's github repository at https://github.com/WestHealth/scipy2020/tree/master/pyvis.

References

[Bev] Andrew Beveridge. Network of thrones. URL: https://www.macalester.edu/~abeverid/thrones.html.

[BH86] J K Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986. doi:10.1038/324446a0.

[Gia18] Giancarlo Perrone. Pyvis interactive network visualizations, 2018. URL: https://pyvis.readthedocs.io/en/latest/.

[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. doi:10.25080/issn.2575-9752.

[Pro] The Pallets Projects. jinja. URL: https://jinja.palletsprojects.com/.

[vis20a] vis.js community. Network - physics, 2020. URL: https://visjs.github.io/vis-network/docs/network/physics.html.

[vis20b] vis.js community. vis.js, 2020. URL: https://visjs.org.

Boost-histogram: High-Performance Histograms as Objects

Henry Schreiner[‡*], Hans Dembinski[§], Shuo Liu[¶], Jim Pivarski[‡]

https://youtu.be/ERraTfHkPd0

Abstract—Unlike arrays and tables, histograms in Python have usually been denied their own object, and have been represented as a single operation producing several arrays. Boost-histogram is a new Python library that provides histograms that can be filled, manipulated, sliced, and projected as objects. Building on top of the Boost libraries' Histogram in C++14 provided interesting distribution and design challenges with useful solutions. This is meant to be a foundation that others can build on; in the Scikit-HEP project[1], a physicist friendly front-end "Hist" and a conversion package "Aghast" are already being designed around boost-histogram.

Index Terms—Histogram, Analysis, Data processing, Data reduction, NumPy, Aggregation

Motivation

As an example of a problem that becomes much easier with histograms as objects, let's look at the Python 3 adoption of several libraries using PyPI download statistics. There are three columns of interest: The package name, the date of the download, and the Python version used when downloading the package. In order to look at trends, you will want to answer questions about the download behavior over time ranges, such as what is the fraction of Python 2 downloads out of all downloads for each month. Let's look at what a solution to this would entail using traditional histogramming methods [NumPy]:

- *Date*: You could make a histogram over datetime objects, but then you will be responsible for finding the bin range (dates are just large numbers), probably using `np.searchsorted` on the edges array, and then making slices in the binned array yourself.
- *Python version*: You would have to force some sort of artificial binning scheme, such as one with edges `[2, 3.0, 3.59, 3.69, 3.79, 4]`, in order to collect information for each Python version of interest. You would have to use a 2D array, and keep the selections/edges straight yourself; in practice, you would probably just create a Python dict of 1D histograms for each major version.
- *Package names*: This would require making a dict and storing each 2D (or set of 1D) histograms manually. NumPy does not support category axes or strings.

If your data doesn't fit into memory, you will have to build in the batching and combining yourself. For each piece.

Now look at this with an object-based Histogram library, such as boost-histogram:

- *Package names*: This can be string categories.
- *Python version*: You could simply multiply by 10 and make these int categories, or just use string categories.
- *Date*: Use a regular spaced binning from start to stop in the resolution you are interested, such as months. Use the loc indexer to convert when slicing. No manual tracking or searching. Use rebinning to convert months into years in one step.

In the object-based version, you fill once. If your data doesn't fit into memory, just fill in batches. The API for ND histograms is identical to 1D histograms, so you don't have to use different functions or change significant portions of code even if you add a new axes later.

Now let's look at using the object to make a series of plots, with one shown in Figure 1[2]. The code required to make the plot is shown below, with minor formatting details removed.

```
for name in hist.axes[0]:
    fig, ax = plt.subplots()
    ax.set_title(name)
    for vers in hist.axes[1]:
        dhist = hist[bh.loc(name), bh.loc(vers), :]
        (dt,) = d.axes.centers
        xs = mpl.dates.date2num(pd.to_datetime(dt))
        ax.plot_date(xs, dhist, label=f"{vers/10}")
```

Note how all the computation, and the version information is stored in a single histogram object. The datetime centers are accessible after the package and version number are selected. Looping over the categories is trivial. Since the histogram is already filled, there are no other loops over the data to slow down manipulation. We could rebin or set limints or sum over axes cleanly as well.

* *Corresponding author: henryfs@princeton.edu*
‡ *Princeton University*
§ *TU Dortmund*
¶ *Sun Yat-sen University*

1. https://scikit-hep.org

2. Code available at https://github.com/scikit-hep/scikit-hep-orgstats

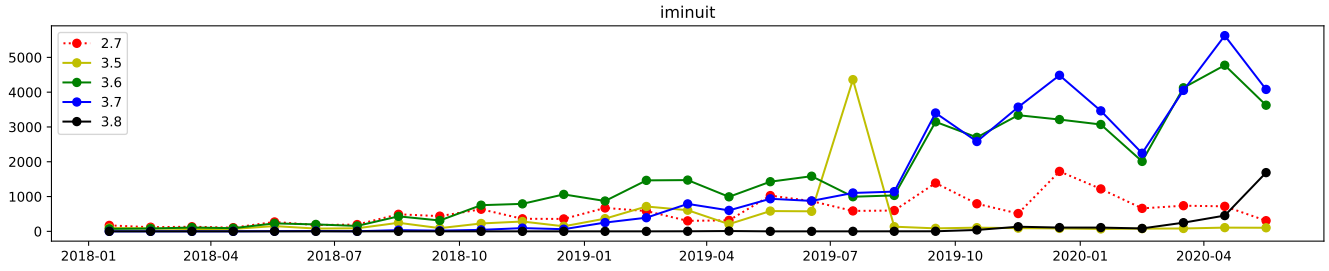


Fig. 1: *A downloads vs. time histogram plot for iMinuit [iMinuit] by Python version, made with Matplotlib [Matplotlib].*

Introduction

In the High Energy Physics (HEP) community, histogramming is vital to most of our analysis. As part of building tools in Python to provide a friendly and powerful alternative to the ROOT C++ analysis stack [ROOT], histogramming was targeted as an area in the Python ecosystem that needed significant improvement. The "histograms are objects" mindset is a general, powerful way of interacting with histograms that can be utilized across disciplines. We have built boost-histogram in cooperation with the Boost C++ community [Boost] for general use, and also have separate more specialized tools built on top of boost-histogram that customize it for HEP analysis (which will be discussed briefly at the end of this paper).

At the start of the project, there were many existing histogram libraries for Python (at least 24 were identified by the authors), but none of them fulfilled the requirements and expectations of users coming from custom C++ analysis tools. Four key areas were identified as key to a good library for creating histograms: Design, Flexibility, Performance, and Distribution.

Before we continue, a brief description of a histogram should suffice to set the stage until we describe boost-histogram's approach in more detail. A histogram reduces an arbitrarily large dataset into a finite set of bins. A histogram consists of one or more *axes* (sometimes called "binnings") that describe a conversion from *data coordinates* to *bin coordinates*. The data coordinates may be continuous or discrete (often called categories); the bin coordinates are always discrete. In NumPy [NumPy], this conversion is internally derived from a combination of the `bin` and `range` arguments. Each *bin* in the histogram stores some sort of aggregate information for each value that falls into it via the axes conversion. This is a simple sum in NumPy. When something besides a sum is used, this is a "generalized histogram", which is called a `binned_statistic` in `scipy.stats` [SciPy]; for our purposes, we will avoid this distinction for the sake of brevity, but our histogram definition does include generalized histograms. Histograms often have an extra "weight" value that is available to this aggregate (a weighted sum in NumPy).

Almost as important as defining what a histogram is limiting what a histogram is not. Notice the missing item above: a histogram, in this definition, is not a plot or a visual aid. It is not a plot any more than a NumPy array is a plot. You can plot a Histogram, certainly, and customisations for plotting are useful (much as Pandas has custom plotting for Series [Pandas]), but that should not part of a core histogram library, and is not part of boost-histogram (though most tutorials include how to plot using Matplotlib [Matplotlib]).

The first area identified was **Design**; here many popular libraries fell short. Histograms need to be represented as an object, rather than a collection of NumPy arrays, in order to naturally manipulate histograms after filling. You should be able to continue to fill a histogram after creating it as well; filling in one pass is not always possible due to memory limits or live data taking conditions. Once a histogram is filled, it should be possible to perform common operations on it, such as rebinning to a courser binning scheme, projecting on a subset of axes, selecting a subset of bins then working with or summing over just that piece, and more. You should be able easily sum histograms, such as from different threads. You also should be able to easily access the transform between data coordinates and bin coordinates for each axes. Axis should be able to store extra information, such as a title or label of some sort, to assist the user and external plotting tools.

The second area identified was **Flexibility**; there are a wide range of things a histogram should be able to do; these traditionally are split into different functions and objects, but as we show, a clear, consistent design makes it possible to unify around a single object. Axes should support several forms of binning: variable width binnings, regularly spaced binnings (a performance-optimized subset of variable binning), and categorical binning. Out-of-range bins (called flow bins, discussed later) are also key for enabling lossless sums over a partial collection of axes. Axes should also be able to optionally grow when a fill is out of range instead. The bins themselves should support simple sums, like NumPy, but should also support means (sometimes called profile histograms). High-precision weighted summing is also useful. Finally, if you add a sample parameter to the fill, you can also keep track of the variance for each bin.

The third area identified was **Performance**; when dealing with very large datasets that will not fit in memory, the filling performance becomes critical. High performance filling is also useful in real-time applications. A highly performance histogram library should support fast filling with a compiled loop, it should avoid reverting to a slower $\mathcal{O}(n)$ lookup when filling a regularly spaced axes, and it should be able to take advantage of multiple cores when filling from a large dataset. NumPy, for example, does do well for a single regularly spaced axes, but it still does not optimize for two regularly spaced axes (an image is an example of a common regularly spaced 2D histogram).

The fourth and final area identified was **Distribution**. A great library is not useful if no one can install it; it is especially important that students and inexperienced users be able to install the histogramming package. This is one of Python's strengths compared to something like C++, but the above requirements necessitate compiled components, so this is important to get right. It also needed to work flawlessly in virtual environments and in the Conda package manager. It also needed to be available on as many platforms and for as many Python versions as possible to

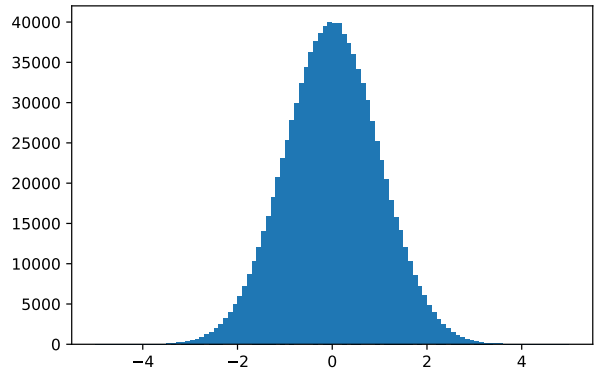


Fig. 2: *An example of a 1D-histogram.*

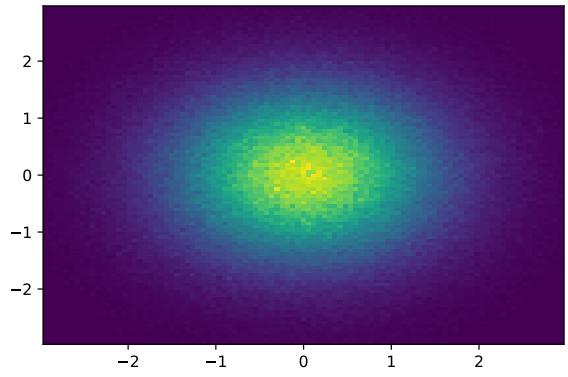


Fig. 3: *An example of a 2D-histogram.*

support both old and new data acquisition and analysis systems.

About a year ago, a new C++14 library was proposed to the Boost C++ libraries called Boost.Histogram; it was unanimously accepted and released as part of the Boost C++ libraries version 1.70 after the review process. It was a well designed header-only package that fulfilled exactly what we wanted, but in C++14 rather than Python. A proposal was made to get a full-featured Python binding developed as part of an institute for sustainable software for HEP [IRIS-HEP], as one of the foundations for a Python based software stack being designed to be part of the Scikit-HEP community [SkHEP]. We built boost-histogram for Python in close collaboration with the original Histogram for Boost author, Hans Dembinski, who had always intended Boost.Histogram to be accessible from Python. Due to this close collaboration, concepts and design closely mimic the spirit of the Boost counterpart.

An example of the boost-histogram library approach, creating a 1D-histogram and adding values, is shown below, with results plotted in Figure 2:

```
import boost_histogram as bh
import numpy as np
import matplotlib.pyplot as plt

ax = bh.axes.Regular(100, start=-5, stop=5)
hist = bh.Histogram(ax)

hist.fill(np.random.randn(1_000_000))

plt.bar(hist.axes[0].centers,
        hist.view(),
        width=hist.axes[0].widths)
```

For future code snippets, the imports used above will be assumed. Using `.view()` is optional, but is included to make these explicit. You can access `ax` as `hist.axes[0]`. Note that boost-histogram is not plotting; this is simply accessing histogram properties and leveraging existing Matplotlib functionality. A similar example, but this time in 2D, is shown in Figure 3, illustrating the identical API regardless of the number of dimensions:

```
hist_2d = bh.Histogram(bh.axis.Regular(100, -3, 3),
                       bh.axis.Regular(100, -3, 3))

hist_2d.fill(np.random.randn(1_000_000),
             np.random.randn(1_000_000))

X, Y = hist_2d.axes.centers
plt.pcolormesh(X.T, Y.T, hist_2d.view().T)
```

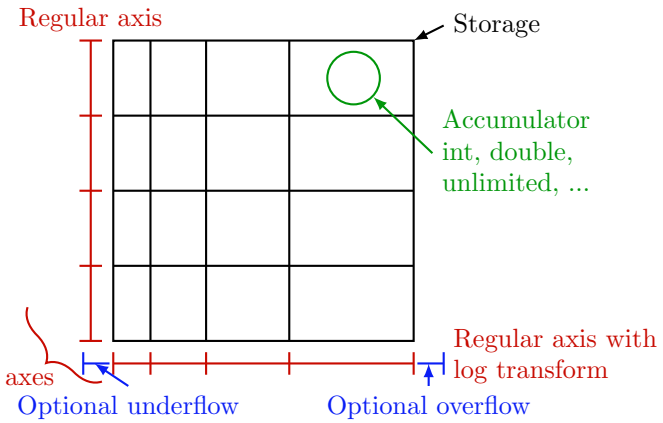


Fig. 4: *The components of a histogram, shown for a 2D histogram.*

Boost-histogram is available on PyPI and conda-forge, and the source is BSD licensed and available on GitHub[3]. Extensive documentation is available on ReadTheDocs[4].

The Design of a Histogram

Let's revisit our description of a histogram, this time mapping boost-histogram components to each piece. See Figure 4 for an example of how these visually fit together to create an 2D histogram.

The components in a bin are the smallest atomic piece of boost-histogram, and are called **Accumulators**. Four such accumulators are available. `Sum` just provides a high-accuracy floating point sum using the Neumaier algorithm [Neu74], and is automatically used for floating point histograms. `WeightedSum` provides an extra term to allow sample sizes to be given. `Mean` stores a mean instead of a sum, created what is sometimes called a "profile histogram". And `WeightedMean` adds an extra term allowing the user to provide samples. Accumulators are like a 0D or scalar histogram, much like dtypes are like 0D scalar arrays in NumPy.

The above accumulators are then provided in a container called a **Storage**, of which boost-histogram provides several. The available storages include choices for the four accumulators listed

3. https://github.com/scikit-hep/boost-histogram
4. https://boost-histogram.readthedocs.io

above (the storage using `Sum` is just called `Double()`, and is the default; unlike the other accumulator-based storages it provides a simple NumPy array rather than a specialized record array when viewed). Other storages include `Int64()`, which stores integers directly, `AtomicInt64`, which stores atomic integers, so can be filled from different threads concurrently, and `Unlimited()`. which is a special growing storage that offers a no-overflow guarantee and automatically uses the least possible amount of memory for a dense uniform array of counters, which is very helpful for high-dimensional histograms. It also automatically converts to doubles if filled with a weighted fill or scaled by a float.

The next piece of a histogram is an **Axis**. A `Regular` axis describes an evenly spaced binning with start and end points, and takes advantage of the simplicity of the transform to provide $\mathcal{O}(1)$ computational complexity. You can also provide a **Transform** for a `Regular` axes; this is a pair of C function pointers (possibly generated by a JIT compiler [Numba]) that can apply a function to the transform, allowing for things like log-scale axes to be supported at the same sort of complexity as a `Regular` axis. Several common transforms are supplied, including log and power spacings. You can also supply a list of bin edges with a `Variable` axis. If you want discrete axes, `Integer` provides a slightly simpler version of a `Regular` axes, and `IntCategory`/`StrCategory` provide true non-continuous categorical axes for arbitrary integers or strings, respectively. Most axes have configurable end behaviors for when a value is encountered by a fill that is outside the range described by the axis, allowing underflow/overflow bins to be turned off, or replaced with growing bins. All axes also have a metadata slot that can store arbitrary Python objects for each axis; no special meaning is applied by boost-histogram, but these can be used for titles, units, or other information.

An example of a custom transform applied to a `Regular` axis is shown below using Numba to create C pointers; any ctypes pointer is accepted.

```
import numba

@numba.cfunc(numba.float64(numba.float64))
def exp(x):
    return math.exp(x)

@numba.cfunc(numba.float64(numba.float64))
def log(x):
    return math.log(x)

transform_log = bh.axis.transform.Function(log, exp)

bh.axis.Regular(10, 1, 4, transform=transform_log)
```

You need to provide both directions in the transform, so that boost-histogram can add values to bins and find bin edges. Note: don't actually use exactly this code; there is a `bh.axis.transform.log` already compiled in the library.

A **Histogram** is the combination of a storage and one or more axes. Histograms always manage their own memory, though they provide a view of that storage to Python via the buffer protocol and NumPy. Histograms have the same API regardless of whether they have one axes or thirty-two, and they have a rich set of interactions defined, which will be the topic of the next section. This is an incredibly flexible design; you can orthogonally combine any mixture of axes and storages with associated accumulators, and in the future, new axes types or accumulators and storages can be added.

Interactions with a Histogram

A Histogram supports a variety of operations, many of which use Python's syntax to be expressed naturally and succinctly. Histograms can be added, copied, pickled (special attention was paid to ensure even accumulator storages are pickled quickly and efficiently), and used most places a NumPy array is accepted. Scaling a histogram can be done simply by using Python's multiplication and division operators.

Conversion to a NumPy array was carefully designed to provide a comfortable interface for Python users. The "flow" bins, which are the bins that are used when an event is encountered outside the range of the current axis, are an essential feature for partial summations. These extra bins are not as common in NumPy based analyses (though you can create flow bins manually in NumPy by using $\pm\infty$), so these generally are not needed or expected when converting to an array. The array interface and all external methods do not include flow bins by default, but they can be activated by passing `flow=True` to any of the methods that could be affected by flow bins. You can directly access a view of the data without flow bins with `.view()`, and you can include flow bins with `.view(flow=True)`. The stride system is descriptive enough to avoid needing to copy memory in either case. Views of accumulator storages are NumPy record arrays, enhanced with property-based access for the fields as well as common computed properties, like the variance. Finally, there is an explicit `.to_numpy()` method that returns the same tuple you would get if you used one of the `np.histogram` functions.

Axes are presented as a property returning an enhanced tuple. You can use access any method or property on all axes at once directly from the `AxesTuple`. Array properties (like edges) are returned in a shape that is ready for broadcasting, allowing natural manipulations directly on the returned values. For example, the following snippet computes the density of a histogram, regardless of the number of dimensions:

```
# Compute the "volume" of each bin (useful for 2D+)
volumes = np.prod(hist.axes.widths, axis=0)

# Compute the density of each bin
density = hist.view() / hist.sum() / volumes
```

Unified Histogram Indexing

Indexing in boost-histogram, based on a proposal called Unified Histogram Indexing (UHI)[5], allows NumPy-like slicing and is based on tags that can be used cross-library. They can be used to select items from axes, sum over axes, and slice as well, in either data or bin coordinates. One of the benefits of the axes based design is that selections that traditionally would have required multiple histograms now can simply be represented as an axes in a single histogram and then UHI is used to select the subset of interest.

The key design is that any indexing expression valid in both NumPy and boost-histogram should return the same thing regardless of whether you have converted the histogram into an array via `.view()` or `np.asarray` or not. Freedom to access the unique parts of boost-histogram are only granted through syntax that is not valid on a NumPy array. This is done through special tags that are not valid in NumPy indexing. These tags do not depend on the internals of boost-histogram, however, and could be written

5. https://boost-histogram.readthedocs.io/en/latest/usage/indexing.html

by a user or come from a different library; the are mostly simple callables, with minor additions to make their *repr*'s look nicer.

There are several tags provided: `bh.loc(float)` converts a data-coordinate into bin coordinates, and supports addition/subtraction. For example, `hist[bh.loc(2.0) + 2]` would find the bin number containing 2.0, then add two to it. There are also `bh.underflow` and `bh.overflow` tags for accessing the flow bins.

Slicing is supported, and works much like NumPy, though it does return a new Histogram object. You can use tags when slicing. A single value, when mixed with a slice, will select out a single value from the axes and remove it, just like it would in NumPy (you will see later why this is very useful). Most interesting, though, is the third parameter of a slice - normally called the step. Stepping in histograms is not supported, as that would be a set of non-continuous but non-discrete bins; but you can pass two different types of tags in. The first is a "rebinning" tag, which can modify the axis -- `bh.rebin(2)` would double the size of the bins. The second is a reduction, of which `bh.sum` is provided; this reduces the bins along an axes to a scalar and removes the axes; `builtins.sum` will trigger this behavior as well. User provided functions will eventually work here, as well. Endpoints on these special operations are important; leaving off the endpoints will include the flow bins, including the endpoints will remove the flow bins. So `hist[::sum]` will sum over the entire histogram, including the flow bins, and `hist[0:len:sum]` will sum over the contents of the histogram, not including the flow bin. Note that Python's *len* is a perfectly valid in this system - start and stop tags are simply callables that accept an axis and return an index from `-1` (underflow bin) to `len(axis)+1` (overflow bin), and axes support `len()`.

Setting is also supported, and comes with one more nice feature. When you set a histogram with an array and one or more endpoints are empty and include a flow bin, you have two options; you can either match the inner size, which will leave the flow bin(s) alone, or you can match the total size, which will fill the flow bins too. For example, in the following snippet the array can be either size 10 or size 12:

```
hist = bh.Histogram(bh.axis.Regular(10, 0, 1))
hist[:] = np.arange(10) # Fills regular bins
hist[:] = np.arange(12) # Fills flow bins too
```

You can force the flow bins to be explicitly excluded if you want to by adding endpoints to the slice:

```
hist[0:len] = np.arange(10)
```

Finally, for advanced indexing, dictionaries are supported, where the key is the axis number. This allows easy access into a large number of axes, or simple programmatic access. With dictionary-based indexing, Ellipsis are not required. There is also a `.project(*axes)` method, which allows you to sum over all axes except the ones listed, which is the inverse to listing `::sum` operations on the axes you want to remove.

Performance when Filling

A histogram can be viewed as a lossy data compression tool; you lose the exact details of each data point, but you have a have a representation that does not depend on the number of data points and has several very useful properties for computation. One common use beyond plotting is distribution fitting; you can fit an arbitrarily large number of data points to a distribution as long as you choose a binning dense enough to capture the details of your distribution function. The performance of the fit is based on the number of bins, rather than the number of measurements made. Many distribution fitting packages available outside of HEP, such as lmfit [LMFIT], are designed to work with binned data, and binned fits are common in HEP as well.

Setup	Single threaded	X	Multithreaded	X
NumPy 1D	74.5 ± 2.4 ms	1		
BH 1D	41.6 ± 0.7 ms	1.8	13.3 ± 0.2 ms	5.5
BHNP 1D	43.1 ± 0.8 ms	1.7	13.8 ± 0.2 ms	5.4
NumPy 2D	874 ± 22 ms	1		
BH 2D	77.6 ± 0.6 ms	11	28.7 ± 0.7 ms	30
BHNP 2D	85 ± 3 ms	10	29.6 ± 0.5 ms	29

TABLE 1: *Comparison of several filling methods and NumPy. BH stands for boost-histogram object mode (as seen above). BHNP stands for boost-histogram NumPy clone, which provides the same interface as NumPy but powered by Boost.Histogram calculations. Multithreaded was obtained by passing `threads=8` while filling. The X column is a comparison against NumPy. Measurements done on an 8 core 16 MBP, 2.4 GHz, Regular binning, 10M values, 32-bit floats.*

Filling performance was a key design goal for boost-histogram. In Table 1 you can see a comparison of filling methods with NumPy. The first comparison, a 1D histogram, shows a nearly 2x speedup compared to NumPy on a single core. For a 1D `Regular` axes, NumPy has a custom fill routine that takes advantage of the regular binning to avoid an edge lookup. If you use multiple cores, you can get an extra 2x-4x speedup. Note that histogramming is not trivial to parallelize. Internally, boost-histogram is just using simple Python threading and relying on releasing the GIL while it fills multiple histograms; the histograms are then added into your current histogram. The overhead of doing the copy must be small compared to the fill being done.

If we move down the table to the 2D case, you will see Boost-histogram pull away from NumPy's 2D regular bin edge lookup with an over 10x speedup. This can be further improved to about 30x using threads. In both cases, boost-histogram is not actually providing specialized code for the 1D or 2D cases; it is the same variadic vector that it would use for any number and any mixture of axes. So you can expect excellent performance that scales well with the complexity of your problem.

The rows labeled "BHNP" deserve special mention. A special module is provided, *bh.numpy*, that contains functions that exactly mimic the functions in NumPy. They even use a special, internal axes type that mimics NumPy's special handling of the final upper edge, including it in the final bin. You can use it as a drop-in replacement for the histogram functions in NumPy, and take advantage of the performance boost available. You can also add the `threads=` keyword. You can pass `histogram=bh.Histogram` to return a Histogram object, and you can select the storage with `storage=`, as well. Combined with the ability to convert Histograms via `.to_numpy()`, this should enable smooth transitions between boost-histogram and NumPy for Histogram filling.

One further performance benefit comes from the flexibility of combining axes. In a traditional, NumPy based analysis, you may have a collection of related histograms with different cuts or criteria for filling. We have already seen that it is possible to use axis and then access the portion you want later with indexing; but

if you have categories or boolean selectors, you can still combine multiple histograms into one. Then you no longer loop over the input multiple times, but just once, filling the histogram, and then make your selections later. Here is an example:

```
value_ax = bh.axis.Regular(100, -5, 5)
valid_ax = bh.axis.Integer(0, 2,
                           underflow=False,
                           overflow=False)
label_ax = bh.axis.StrCategory([], growth=True)

hist = bh.Histogram(value_ax, valid_ax, label_ax)

hist.fill([-2, 2, 4, 3],
          [True, False, True, True],
          ["a", "b", "a", "b"])

all_valid = hist[:, bh.loc(True), ::sum]
a_only = hist[..., bh.loc("a")]
```

Above, we create three axes. The second axis is a boolean axes, which hold a valid/invalid bool flag. The third axis holds some sort of string-based category, which could label datasets, for example. We then fill this in one shot. Then, we can select the histograms that we might have originally filled separately, like the `all_valid` histogram, which is a 1D histogram that contains all labels and all events where `valid=True`. In the second selection, `a_only`, a 2D histogram is returned that consists of all the events labeled with `"a"`.

This way of thinking can radically change how you design for a problem. Instead of running a series of histograms over a piece of data every time you want a new selection, you can build a large histogram that contains all the information you want, prebinned and ready to select. This combination of multiple histograms and later selecting or summing along axes is a close parallel to the way Pandas combines multiple NumPy arrays in a single DataFrame using columns, allowing you to group and select from the full set.

Distributing

Building a Python library designed to work absolutely anywhere on a C++14 code base provided several challenges. Binding for boost-histogram is accomplished with PyBind11 [PyBind], and all Boost dependencies are included via git submodules and header-only, so a compatible compiler is the only requirement for building if a binary is not available. Serialization, which optionally depends on the non-header only Boost.Serialization, was redesigned to work on top of Python tuple picking in PyBind11 reusing the same interface internally in Boost.Histogram (one of the many benefits of a close collaboration with the original author).

The first phase of wheel building was a custom set of shareable YAML template files for Azure DevOps. This tool, azure-wheel-helpers[6], became the basis for building several other projects in Scikit-HEP, including the iMinuit fitter[7] and the new Awkward 1.0 [Awkward]. Building a custom wheel production from scratch is somewhat involved; and since boost-histogram is expected to support Python 2.7 until after the first LTS release, it had to include Python 2.7 builds, which make the process even more convoluted. To get C++14 support in manylinux1, a custom docker repository (`skhep/manylinuxgcc`[8]) was developed with GCC 9. The azure-wheel-helpers repository is a good place to look for anyone wishing to learn about wheel building, but recently boost-histogram moved to a better solution.

As the cibuildwheel [CIBW] project matured, boost-histogram became the first Scikit-HEP azure-wheel-helpers project to migrate over. Several of the special cases that were originally supported in boost-histogram are now supported by cibuildwheel, and it allows a custom docker image, so the modified manylinux1 image is available as well. This has freed us from lock-in to a particular CI provider; boost-histogram now uses GitHub Actions for everything except ARM and Power PC builds, which are done on Travis CI. This greatly simplified the release process. The scikit-hep.org developer pages now have extensive tutorials for new developers, including setting up wheels; much of that work was inspired by boost-histogram.

An extremely important resource for HEP is Conda; many of our projects (such as CERN's ROOT toolkit) cannot reasonably (at least yet) be distributed by pip. Scikit-HEP has a large number of packages in conda-forge; and boost-histogram is also available there, including ARM and PowerPC builds. Only Python 2.7 on Windows is excluded due to conda-forge policies on using extra SDKs with Python.

Conclusion and Plans

The future for histogramming in Python is bright. At least three more projects are being developed on top or using boost-histogram. **Hist**[9] is a histogram front-end for analysts, much like Pandas is to NumPy, it is intended to make plotting, statistics, file IO, and more simple and easy; a Google Summer of Code student is working on that. One feature of note is named axes; you can assign names to axes and then fill and index by name. Conversions between histogram libraries, such as the HEP-specific ROOT toolkit and file format are being developed in **Aghast**[10]. The **mplhep**[11] library is making common plot styles and types for HEP easy to make, including plots with histograms. The **scikit-hep-tutorials**[12] project is beginning to show how the different pieces of Scikit-HEP packages work together, and one of the first tutorials shows boost-histogram and Aghast. And a new library, **histoprint**[13], is being reviewed for including in Scikit-HEP to print up to five histograms at a time on the command line, either from ROOT or boost-histogram.

An example of mplhep and boost-histogram interaction is shown in Figure 5:

```
import mplhelp
mplhep.histplot(hist)
```

We hope that more libraries will be interested in building on top of boost-histogram. It was designed to be a powerful back-end for any front-end, with Hist planned as the reference front-end implementation. The high performance, excellent flexibility, and universal availability make an ideal choice for any toolkit.

In conclusion, boost-histogram provides a powerful abstraction for histograms as a collection of axes with an accumulator-backed storage. Filling and manipulating histograms is simple and natural, while being highly performant. In the future, Scikit-HEP is rapidly building on this foundation and we expect other libraries may want to build on this as well. At the same time, Boost.Histogram in C++ is continuously improved and expanded with new features,

6. https://github.com/scikit-hep/azure-wheel-helpers
7. https://github.com/scikit-hep/iminuit
8. https://github.com/scikit-hep/manylinuxgcc
9. https://github.com/scikit-hep/hist
10. https://github.com/scikit-hep/aghast
11. https://github.com/scikit-hep/mplhep
12. https://github.com/scikit-hep/scikit-hep-tutorials
13. https://github.com/scikit-hep/histoprint

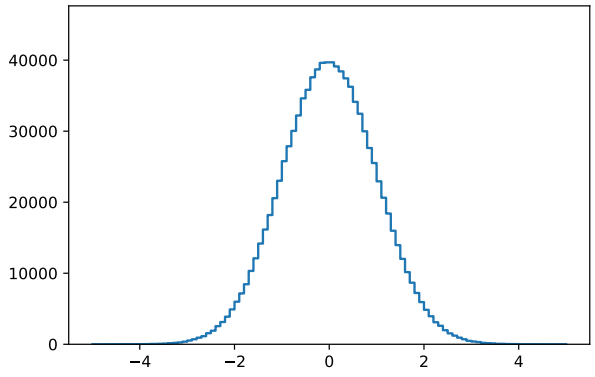


Fig. 5: *An example of a 1D plot with mplhep. It is not completely trivial to get a proper "skyline" histogram plot from Matplotlib with prebinned data, while here it is simple.*

from which boost-histogram benefits nearly automatically. The shared code-base with C++ allows Python to profit, while boost-histogram in C++ is profiting from ideas feed back from Python, creating a win-win situation for all parties.

Acknowledgements

Support for this work was provided by the National Science Foundation cooperative agreement OAC-1836650 (IRIS-HEP) and OAC-1450377 (DIANA/HEP).

REFERENCES

[Pandas] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010), DOI:10.25080/Majora-92bf1922-00a

[NumPy] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, vol. 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[iMinuit] *iminuit — A Python interface to Minuit*, https://github.com/scikit-hep/iminuit

[Matplotlib] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing in Science & Engineering, vol. 9, no. 3, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[ROOT] Rene Brun and Fons Rademakers *ROOT — An Object Oriented Data Analysis Framework* Nucl. Inst. & Meth. A, vol. 386, no. 1, 81-86 (1997), DOI:10.1016/S0168-9002(97)00048-X

[Boost] *The Boost Software Libraries*, https://www.boost.org

[SciPy] Pauli Virtanen et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods, in press. DOI:10.1038/s41592-019-0686-2

[IRIS-HEP] *Institute for Research and Innovation in Software for High Energy Physics*, https://iris-hep.org

[SkHEP] Eduardo Rodrigues. *The Scikit-HEP Project*, EPJ Web Conf. **214** 06005 (2019), DOI:10.1051/epjconf/201921406005

[Neu74] A. Neumaier. *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, Zeitschrift für Angewandte Mathematik und Mechanik (1974), DOI:10.1002/zamm.19740540106

[Numba] Siu Kwan Lam, Antoine Pitrou, Stanley Seibert. *Numba: a LLVM-based Python JIT compiler*, LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 7, 1-6 (2015), DOI:10.1145/2833157.2833162

[LMFIT] Matthew Newville et al. *LMFIT: Non-Linear Least-Square Minimization and Curve-Fitting for Python*, Zenodo (2020), DOI:10.5281/zenodo.3814709

[PyBind] Wenzel Jakob, Jason Rhinelander, Dean Moldovan. *pybind11 -- Seamless operability between C++11 and Python*, https://github.com/pybind/pybind11

[Awkward] Jim Pivarski, Peter Elmer, David Lange. *Awkward Arrays in Python, C++, and Numba* Preprint arXiv:2001.06307

[CIBW] Joe Rickerby et al. *cibuildwheel*, https://github.com/joerick/cibuildwheel

Learning from evolving data streams

Jacob Montiel[‡*]

https://youtu.be/sw85SCv847Y

Abstract—Ubiquitous data poses challenges on current machine learning systems to store, handle and analyze data at scale. Traditionally, this task is tackled by dividing the data into (large) batches. Models are trained on a data batch and then used to obtain predictions. As new data becomes available, new models are created which may contain previous data or not. This training-testing cycle is repeated continuously. Stream learning is an active field where the goal is to learn from infinite data streams. This gives rise to additional challenges to those found in the traditional batch setting: First, data is not stored (it is infinite), thus models are exposed only once to single samples of the data, and once processed those samples are not seen again. Models shall be ready to provide predictions at any time. Compute resources such as memory and time are limited, consequently, they shall be carefully managed. The data can drift over time and models shall be able to adapt accordingly. This is a key difference with respect to batch learning, where data is assumed static and models will fail in the presence of change. Model degradation is a side-effect of batch learning in many real-world applications requiring additional efforts to address it. This papers provides a brief overview of the core concepts of machine learning for data streams and describes scikit-multiflow, an open-source Python library specifically created for machine learning on data streams. scikit-multiflow is built to serve two main purposes: easy to design and run experiments, easy to extend and modify existing methods.

Index Terms—machine learning, data streams, concept drift, scikit, open-source

Introduction

The minimum pipeline in machine learning is composed of: (1) data collection and processing, (2) model training, and (3) model deployment. Conventionally, data is collected and processed in batches. Although this approach is state-of-the-art in multiple applications, it is not suitable in the context of evolving data streams. The batch learning approach assumes that data is sufficiently large and accessible. This is not the case in streaming data where data is available one sample at a time, and storing it is impractical given its (theoretically) infinite nature. In addition, non-stationary environments require to run the pipeline multiple times in order to minimize model degradation, in other words maintain optimal performance. This is especially challenging in fast-changing environments where efficient and effective adaptation is vital.

As a matter of fact, multiple real-world machine learning applications exhibit the characteristics of evolving data streams, in particular we can mention:

- Financial markets generate huge volumes of data daily. For instance, the New York Stock Exchange captures 1 terabyte of information each day[1]. Depending on the state of such markets and multiple external factors data can become obsolete quickly rendering it useless for creating accurate models. Predictive models must be able to adapt fast to be useful in this dynamic environment.
- Predictive maintenance. The contribution of IoT to the digital universe is substantial. Data only from embedded systems accounted for 2% of the world's data in 2013, and is expected to hit 10% by 2020[2]. IoT sensors are used to monitor the health of multiple systems, from complex systems such as airplanes to simpler ones such as household appliances. Predictive systems are required to react fast to prevent disruptions from malfunctioning elements.
- Online fraud detection. The speed of reaction of an automatic system is also an important factor in multiple applications. As a case in point, VisaNet has a capacity (as of June 2019) to handle more than 65,000 transactions per second[3]. Fraud detection in online banking involves additional challenges beside data collection and processing. Fraud detection systems must adapt quickly to changes such as consumer behavior (for example during holidays), the stability of the financial markets, as well as the fact that attackers constantly change their behavior to beat these systems.
- Supply chain. Several sectors use automatic systems in their supply chain to cope with the demand for products efficiently. However, the COVID-19 pandemic brought to attention the fragility of these systems to sudden changes, e.g., in less than 1 week, products related to the pandemic such as face masks filled the top 10 searched terms in Amazon[4]. Many automatic systems failed to cope with change resulting in the disruption in the supply chain.
- Climate change. Environmental science data is a quintessential example of the five *v*'s of big data: volume, velocity, variety, veracity, and value. In particular, NASA's Earth Science Data and Information System project, holds 24 petabytes of data in its archive and distributed 1.3 billion files in 2017[5]. Understanding environmental data has many implications in our daily lives, e.g., food production can be severally impacted by climate change, disruption of the water cycle has resulted in a rise of heavy rains with the associated risk of floodings. IoT sensors are now making environmental data available at a faster rate and

* *Corresponding author: jacob.montiel@waikato.ac.nz*
‡ *Department of Computer Science, University of Waikato*

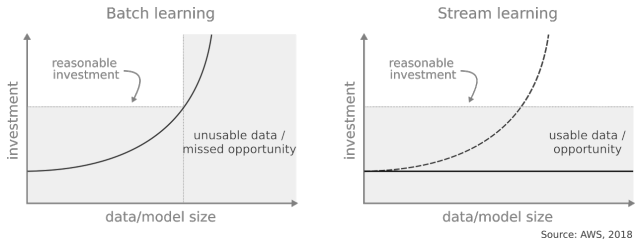


Fig. 1: *Batch learning systems are characterized by the investment in resources like memory and training time as the volume of data increases. Once a* reasonable investment threshold *is reached, data becomes unusable turning into a missed opportunity. On the other hand, efficient management of resources makes stream learning an interesting alternative for big data applications.*

machine learning systems must adapt to this new norm.

As shown in the previous examples, dynamic environments pose an additional set of challenges to batch learning systems. Model degradation is a predominant problem in multiple real-world applications. As enough data has been generated and collected, proactive users might decide to train their models to make sure that they agree with the current data. This is complicated for two reasons: First, batch models (in general) are not able to use new data into account, so the machine learning pipeline must be run multiples times as data is collected over time. Second, the decision for such an action is not trivial and involves multiple aspects. For example, should a new model be trained only on new data? This depends on the amount of variation in the data. Small variations might not be enough to justify retraining and re-deploying a model. This is why a reactive approach is predominantly employed in the industry. Model degradation is monitored and corrective measures are enforced if a user-defined threshold is exceeded (accuracy, type I, and type II errors, etc.). Fig. 1 depicts another important aspect to consider, the tradeoff between the investment in resources such as memory and time (and associated cost) and the pay-off in predictive performance. In stream learning, resource-wise efficiency is fundamental, predictive models not only must be accurate but also must be able to handle theoretically infinite data streams. Models must fit in memory no matter the amount of data seen (constant memory). Additionally, training time is expected to grow sub-linearly with respect to the volume of data processed. New samples must be processed as soon as they become available so it is vital to process them as fast as possible to be ready for the next sample in the stream.

Machine learning for streaming data

Formally, the task of supervised learning from evolving data streams is defined as follows. Consider a stream of data $S = \{(\vec{x}_t, y_t)\} | t = 1, \ldots, T$ where $T \to \infty$. Input $\vec{x}_t$ is a feature vector and y_t the corresponding target where y is continuous in the case of regression and discrete for classification. The objective is to predict the target $\hat{y}$ for an unknown sample $\vec{x}$. For illustrative purposes, this paper focuses on the classification task.

In stream learning, models are trained incrementally, one sample at a time, as new samples $(\vec{x}_t, y_t)$ become available. Since streams are theoretically infinite, the training phase is non-stop and predictive models are continuously updating their internal state in agreement with incoming data. This is fundamentally different from the batch learning approach, where models have access to all (available) data during training. As previously mentioned, in the stream learning paradigm, predictive models must be resource-wise efficient. For this purpose, a set of requirements [BHKP11] must be fulfilled by streaming methods:

- **Process one sample at a time, and inspect it only once.** The assumption is that there is not enough time nor space to store multiple samples, failing to meet this requirement implies the risk of missing incoming data.
- **Use a limited amount of memory.** Data streams are assumed infinite, thus storing data for further processing is impractical.
- **Work in a limited amount of time.** In other words, avoid bottlenecks generated by time-consuming tasks which in the long run could make the algorithm fail.
- **Be ready to predict at any point.** Stream models are continuously *updated* and must be able to provide predictions at any point in time.

Concept drift

A challenging element of dynamic environments is the chances that the underlying relationship between features X and target(s) $\vec{y}$ can evolve (change) over time. This phenomenon is known as **Concept Drift**. Real concept drift is defined as changes in the posterior distribution of the data $p(\vec{y}|X)$. Real concept drift means that the unlabeled data distribution does not change, whereas data evolution refers to the unconditional data distribution $p(X)$. In batch learning, the joint distribution of data $p(X, \vec{y})$ is, in general, assumed to remain stationary. In the context of evolving data streams, concept drift is defined between two points in time t_o, t_1 as

$$p_{t_0}(X, \vec{y}) \neq p_{t_1}(X, \vec{y})$$

Concept drift is known to harm learning [GZB+14]. The following patterns, shown in Fig. 2, are usually considered:

- **Abrupt.** When a new concept is immediately introduced. The transition between concepts is minimal. In this case, adaptation time is vital since the old concept becomes is no longer valid.
- **Incremental.** It can be interpreted as the transition from an old concept into a new concept where intermediate concepts appear during the transition.
- **Gradual.** When old and new concepts concur within the transition period. It can be challenging since both concepts are somewhat valid during the transition.
- **Recurring.** If an old concept is seen again as the stream progresses. For example, when the data corresponds to a periodic phenomenon such as the circadian rhythm.
- **Outliers.** Not to be confused with true drift. A drift detection method must be robust to noise, in other words, minimize the number of false positives in the presence of outliers or noise.

1. How Big Data Has Changed Finance, Trevir Nath, Investopedia, June 2019.

2. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, IDC, April 2014.

3. Visa fact sheet, July 2019.

4. Our weird behavior during the pandemic is messing with AI models. Will Douglas Heaven. MIT Technology Review. May 11, 2020.

5. Big data goes green, Neil Savage, Nature Index 2018 Earth and Environmental Sciences, June 2018

Fig. 2: *Drift patterns depicted as the change of mean data values over time. Note that an outlier is not a change but* noise *in the data. This figure is based on [GZB+14].*

Although the continuous learning nature of stream methods provides some robustness to concept drift, specialized methods have been proposed to detect drift. Multiple methods have been proposed in the literature, [GZB+14] provides a thorough survey of this topic. In general, the goal of drift detection methods is to accurately detect changes in the data distribution while showing robustness to noise and being resources-wise efficient. Drift-aware methods use specialized detection mechanisms to react faster and efficiently to drift. For example, the *Hoeffding Tree* algorithm [DH00], a kind of decision tree for data streams, does not handle concept drift explicitly, whereas the *Hoeffding Adaptive Tree* [BG09] uses *ADaptive WINdowing* (*ADWIN*) [BG07] to detect drifts. If a drift is detected at a given branch, an alternate branch is created and eventually replaces the original branch if it shows better performance on new data.

ADWIN, a popular drift detection method with mathematical guarantees, keeps a variable-length window of recent items; guaranteeing that there has been no change in the data distribution within the window. Internally, two sub-windows (W_0, W_1) are used to determine if a change has happened. With each new item observed, the average values of items in W_0 and W_1 are compared to confirm that they correspond to the same distribution. If the distribution equality no longer holds, then an alarm signal is raised indicating that drift has occurred. Upon detecting a drift, W_0 is replaced by W_1 and a new W_1 is initialized.

Performance evaluation

Predictive performance P of a given model h is usually measured using some loss function ℓ that evaluates the difference between expected (true) class labels y and the predicted class labels $\hat{y}$.

$$P(h) = \ell(y, \hat{y})$$

A popular and straightforward loss function for classification is the *zero-one loss function* which corresponds to the notion of whether the model made a mistake or not when predicting.

$$\ell(y, \hat{y}) = \begin{cases} 0, & y = \hat{y} \\ 1, & y \neq \hat{y} \end{cases}$$

Due to the incremental nature of stream leaning methods, special considerations are used to evaluate their performance. Two prevalent methods in the literature are *holdout* [Koh95] and *prequential* [Daw84] evaluation. Holdout evaluation is a popular method in both batch and stream learning where testing is performed on an independent set of samples. On the other hand, prequential evaluation, is specific to the stream setting. In prequential evaluation, tests are performed on new data samples *before* they are used to train (update) the model. The benefit of this approach is that all samples are used for both test and training.

This is just a brief overview of machine learning for streaming data. However, it is important to mention that the field of machine learning for streaming data covers other tasks such as regression, clustering, anomaly detection, to name a few. We direct the reader to [GRB+19] for an extensive and deeper description of this field, the state-of-the-art, and its active challenges.

The scikit-multiflow package

scikit-mutliflow [MRBA18] is a machine learning library for multi-output/multi-label and stream data written in Python. Developed as free and open-source software and distributed under the BSD 3-Clause License. Following the **SciKits** philosophy, scikit-multiflow extends the existing set of tools for scientific purposes. It features a collection of state-of-the-art methods for classification, regression, concept drift detection and anomaly detection, alongside a set of data generators and evaluators. scikit-multiflow is designed to seamlessly interact with *NumPy* [vCV11] and *SciPy* [VGO+20]. Additionally, it contributes to the democratization of stream learning by leveraging the popularity of the Python language. scikit-multiflow is mainly written in Python, and some core elements are written in *Cython* [BBC+11] for performance.

scikit-multiflow is intended for users with different levels of expertise. Its conception and development follow two main objectives:

1) Easy to design and run experiments. This follows the need for a platform that allows fast prototyping and experimentation. Complex experiments can be easily performed using evaluation classes. Different data streams and models can be analyzed and benchmarked under multiple conditions, and the amount of code required from the user is kept to the minimum.
2) Easy to extend existing methods. Advanced users can create new capabilities by extending or modifying existing methods. This way users can focus on the details of their work rather than on the overhead when working from scratch

scikit-multiflow is not intended as a stand-alone solution for machine learning. It integrates with other Python libraries such as *Matplotlib* [Hun07] for plotting, *scikit-learn* [PVG+11] for incremental learning[6] compatible with the streaming setting, *Pandas* [pdt20] for data manipulation, *Numpy* and *SciPy* for numerical and scientific computations. However, it is important to note that scikit-multiflow does not extend *scikit-learn*, whose main focus in on batch learning. A key difference is that estimators in scikit-multiflow are incremental by design and training is performed by calling multiple times the `partial_fit()` method. The majority of estimators implemented in scikit-multiflow are instance-incremental, meaning single instances are used to update their internal state. A small number of estimators are batch-incremental, where mini-batches of data are used. On the other hand, calling `fit()` multiple times on a scikit-learn estimator will result in it overwriting its internal state on each call.

As of version 0.5.0, the following sub-packages are available:

6. Only a small number of methods in scikit-learn are incremental.

- `anomaly_detection`: anomaly detection methods.
- `data`: data stream methods including methods for batch-to-stream conversion and generators.
- `drift_detection`: methods for concept drift detection.
- `evaluation`: evaluation methods for stream learning.
- `lazy`: methods in which generalization of the training data is delayed until a query is received, e.g., neighbors-based methods such as kNN.
- `meta`: meta learning (also known as ensemble) methods.
- `neural_networks`: methods based on neural networks.
- `prototype`: prototype-based learning methods.
- `rules`: rule-based learning methods.
- `transform`: perform data transformations.
- `trees`: tree-based methods,

In a nutshell

In this section, we provide a quick overview of different elements of scikit-multiflow and show how to easily define and run experiments in scikit-multiflow. Specifically, we provide examples of classification and drift detection.

Architecture

Here we describe the basic components of scikit-multiflow. The `BaseSKMObject` class is the base class. All estimators in scikit-multiflow are created by extending the base class and the corresponding task-specific mixin(s): `ClassifierMixin`, `RegressorMixin`, `MetaEstimatorMixin` and `MultiOutputMixin`.

The `ClassifierMixin` defines the following methods:

- `partial_fit` -- Incrementally train the estimator with the provided labeled data.
- `fit` -- Interface used for passing training data as batches. Internally calls `partial_fit`.
- `predict` -- Predict the class-value for the passed unlabeled data.
- `predict_proba` -- Calculates the probability of a sample pertaining to a given class.

During a learning task, three main tasks are performed: data is provided by the stream, the estimator is trained on incoming data, the estimator performance is evaluated. In scikit-multiflow, data is represented by the `Stream` class, where the `next_sample()` method is used to request new data. The `StreamEvaluator` class provides an easy way to set-up experiments. Implementations for holdout and prequential evaluation methods are available. A stream and one or more estimators can be passed to an evaluator.

Classification task

In this example, we will use the SEA generator. A stream generator does not store any data but generates it on demand. The `SEAGenerator` class creates data corresponding to a binary classification problem. The data contains 3 numerical features, from which only 2 are relevant for learning[7]. We will use the data from the generator to train a Naive Bayes classifier. For compactness, the following examples do not include import statements, and external libraries are referenced by standard aliases.

As previously mentioned, a popular method to monitor the performance of stream learning methods is the prequential evaluation. When a new data sample `(X, y)` arrives: 1. Predictions are obtained for the new data sample (X) to evaluate how well the model performs. 2. Then the new data sample `(X, y)` is used to train the model so it updates its internal state. The prequential evaluation can be easily implemented as a loop:

```
stream = SEAGenerator(random_state=1)
classifier = NaiveBayes()

n_samples = 0
correct_cnt = 0
max_samples = 2000

# Prequential evaluation loop
while n_samples < max_samples and \
stream.has_more_samples():
    X, y = stream.next_sample()
    # Predict class for new data
    y_pred = classifier.predict(X)
    if y[0] == y_pred[0]:
        correct_cnt += 1
    # Partially fit (train) model with new data
    classifier.partial_fit(X, y)
    n_samples += 1

print('{} samples analyzed.'.format(n_samples))
print('Accuracy: {}'.format(correct_cnt / n_samples))
```

```
> 2000 samples analyzed.
> NaiveBayes classifier accuracy: 0.9395
```

The previous example shows that the Naive Bayes classifier achieves an accuracy of 93.95% after processing all the samples. However, learning from data streams is a continuous task and a best-practice is to monitor the performance of the model at different points of the stream. In this example, we use an instance of the `Stream` class as it provides the `next_sample()` method to request data and the returned data is a tuple of `numpy.ndarray`. Thus, the above loop can be easily modified to read from other data structures such as `numpy.ndarray` or `pandas.DataFrame`. For real-time applications where data is actually represented as a stream (e.g. Google's protocol buffers), the `Stream` class can be extended to wrap the necessary code to interact with the stream.

The prequential evaluation method is implemented in the `EvaluatePrequential` class. This class provides extra functionalities including:

- Easy setup of different evaluation configurations
- Selection of different performance metrics
- Visualization of performance over time
- Ability to benchmark multiple models concurrently
- Saving evaluation results to a csv file

We can run the same experiment on the SEA data. This time we compare two classifiers: `NaiveBayes` and `SGDClassifier` (linear SVM with SGD training). We use the `SGDClassifier` in order to demonstrate the compatibility with incremental methods from scikit-learn.

```
stream = SEAGenerator(random_state=1)
nb = NaiveBayes()
svm = SGDClassifier()
# Setup the evaluation
metrics = ['accuracy', 'kappa',
           'running_time', 'model_size']
eval = EvaluatePrequential(show_plot=True,
                           max_samples=20000,
                           metrics=metrics)
# Run the evaluation
```

7. Some data generators and estimators use random numbers generators. When set, the `random_state` parameter enforces reproducible results.

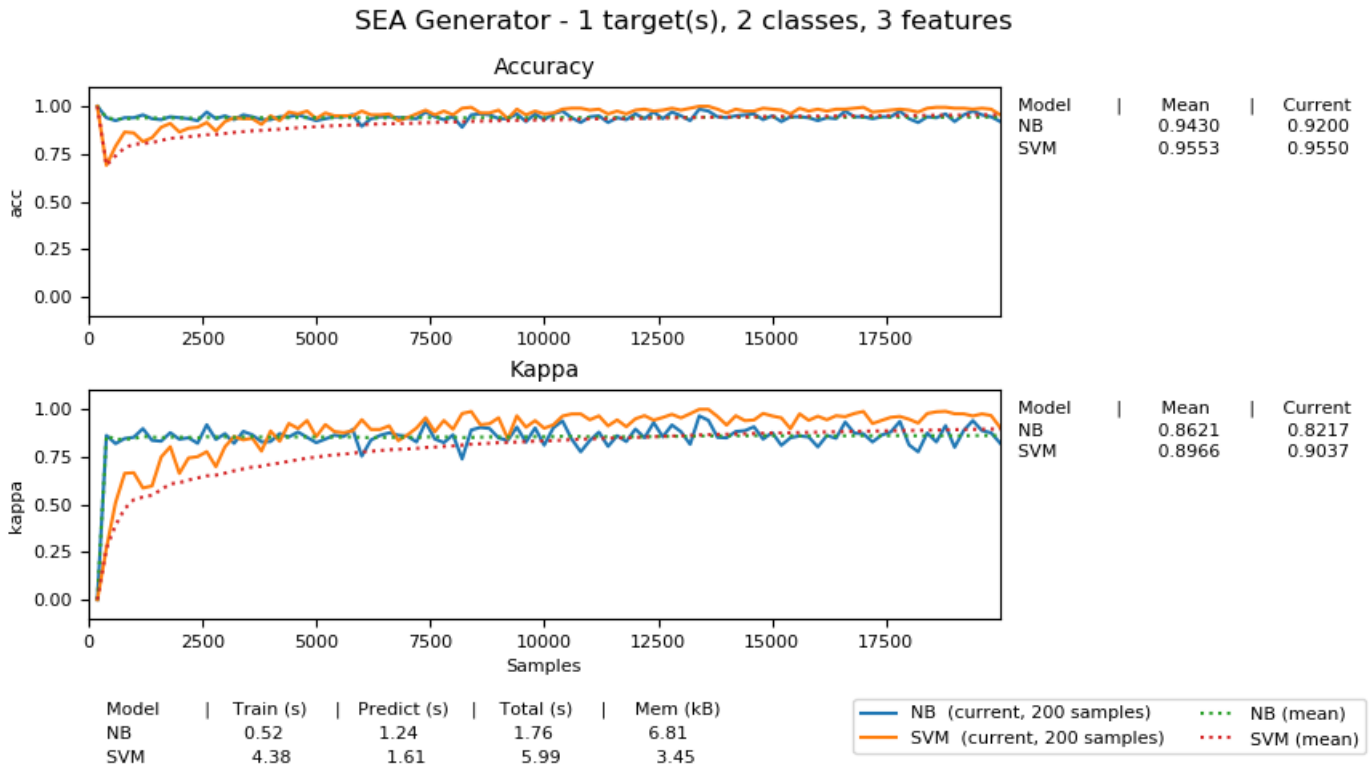


Fig. 3: *Performance comparison between* `NaiveBayes` *and* `SGDClassifier` *using the* `EvaluatePrequential` *class.*

```
eval.evaluate(stream=stream, model=[nb, svm],
                     model_names=['NB', 'SVM']);
```

We set two metrics to measure predictive performance: accuracy and kappa statistics [Coh60] (for benchmarking classification accuracy under class imbalance, compares the models accuracy against that of a random classifier). During the evaluation, a dynamic plot displays the performance of both estimators over the stream, Fig. 3. Once the evaluation is completed, a summary is displayed in the terminal. For this example and considering the evaluation configuration:

```
Processed samples: 20000
Mean performance:
NB - Accuracy     : 0.9430
NB - Kappa        : 0.8621
NB - Training time (s)  : 0.56
NB - Testing time  (s)  : 1.31
NB - Total time    (s)  : 1.87
NB - Size (kB)          : 6.8076
SVM - Accuracy     : 0.9560
SVM - Kappa        : 0.8984
SVM - Training time (s)  : 4.70
SVM - Testing time  (s)  : 1.73
SVM - Total time    (s)  : 6.43
SVM - Size (kB)          : 3.4531
```

In Fig. 3, we observe the evolution of both estimators as they are trained on data from the stream. Although `NaiveBayes` has better performance at the beginning of the stream, `SGDClassifier` eventually outperforms it. In the plot we show performance at multiple points, measured by the given metric (accuracy, kappa, etc.) in two ways: *Mean* corresponds to the average performance on all data seen previously, resulting in a smooth line. *Current* indicates the performance over a sliding window with the latest data from the stream, The size of the sliding window can be defined by the user and is useful to analyze the 'current' performance of an estimator. In this experiment, we also measure resources in terms of time (training + testing) and memory. `NaiveBayes``is faster and uses slightly more memory. On the other hand, ``SGDClassifier` is slower and has a smaller memory footprint.

Concept drift detection

For this example, we will generate a synthetic data stream. The first 1000 samples of the stream contain a sequence from a normal distribution with $\mu_a = 0.8$, $\sigma_a = 0.05$, followed by 1000 samples from a normal distribution with $\mu_b = 0.4$, $\sigma_b = 0.2$, and the last 1000 samples from a normal distribution with $\mu_c = 0.6$, $\sigma_c = 0.1$. The distribution of data in the described synthetic stream is shown in Fig. 4.

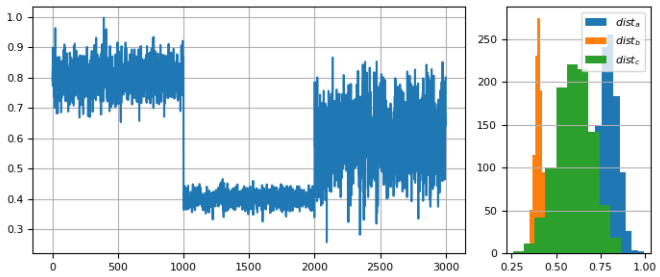


Fig. 4: *Synthetic data simulating a drift. The stream is composed by two distributions of 500 samples.*

```
random_state = np.random.RandomState(12345)
dist_a = random_state.normal(0.8, 0.05, 1000)
```

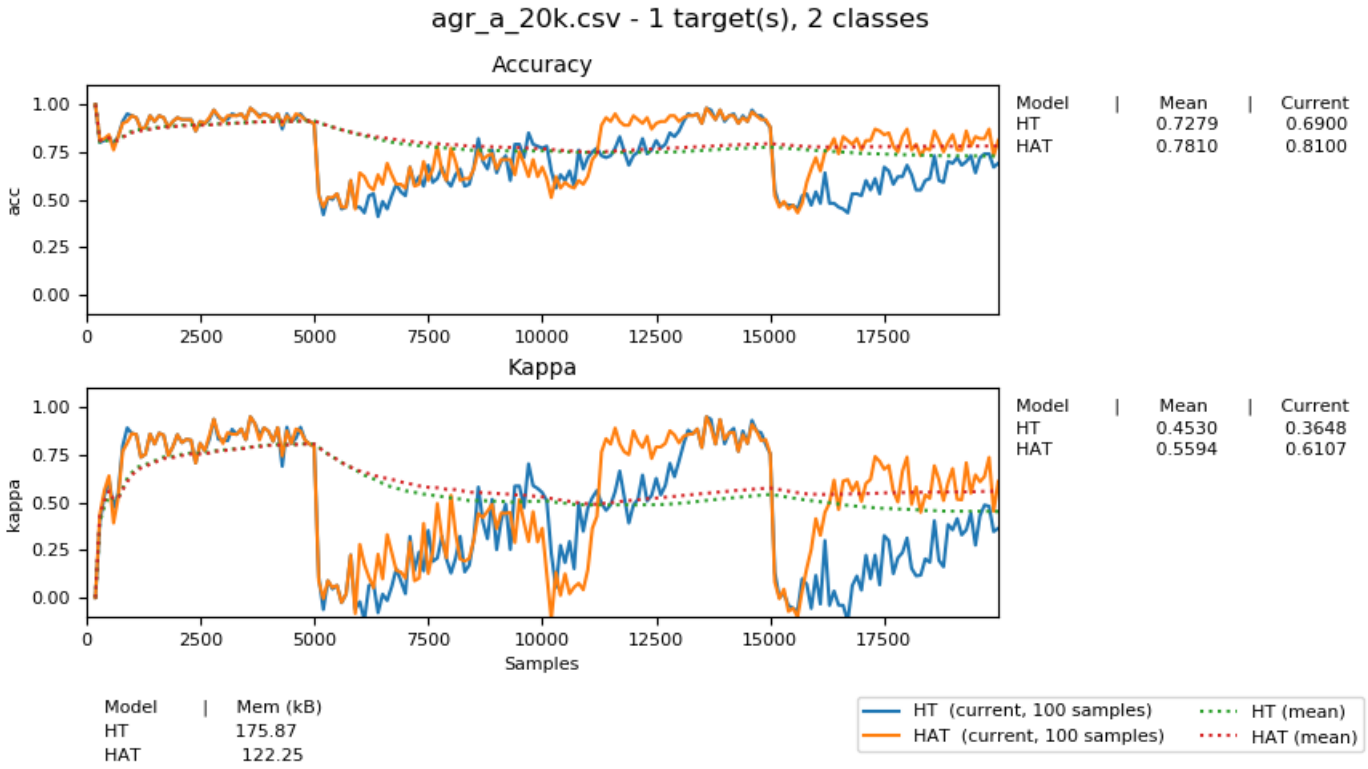


Fig. 5: *Benchmarking the Hoeffding Tree vs the Hoeffding Adaptive Tree on presence of drift.*

```
dist_b = random_state.normal(0.4, 0.02, 1000)
dist_c = random_state.normal(0.6, 0.1, 1000)
stream = np.concatenate((dist_a, dist_b, dist_c))
```

We will use the ADaptive WINdowing (ADWIN) drift detection method. The goal is to detect that drift has occurred after samples 1000 and 2000 in the synthetic data stream.

```
drift_detector = ADWIN()

for i, val in enumerate(stream_int):
    drift_detector.add_element(val)
    if drift_detector.detected_change():
        print('Change detected at index {}'.format(i))

        drift_detector.reset()
```

```
> Change detected at index 1055
> Change detected at index 2079
```

Impact of drift on learning

Concept drift can have a significant impact on predictive performance if not handled properly. Most batch models will fail in the presence of drift as they are essentially trained on different data. On the other hand, stream learning methods continuously update themselves and can adapt to new concepts. Furthermore, drift-aware methods use change detection methods to trigger mitigation mechanisms if a change in performance is detected.

In this example, we compare two popular stream models: the `HoeffdingTreeClassifier`, and its drift-aware version, the `HoeffdingAdaptiveTreeClassifier`.

For this example, we will load the data from a csv file using the `FileStream` class. The data corresponds to the output of the `AGRAWALGenerator` with 3 gradual drifts at the 5k, 10k, and 15k marks. A gradual drift means that the old concept is gradually replaced by a new one, in other words, there exists a transition period in which the two concepts are present.

```
stream = FileStream("agr_a_20k.csv")
ht = HoeffdingTreeClassifier(),
hat = HoeffdingAdaptiveTreeClassifier()
# Setup the evaluation
metrics = ['accuracy', 'kappa', 'model_size']
eval = EvaluatePrequential(show_plot=True,
                           metrics=metrics,
                           n_wait=100)
# Run the evaluation
eval.evaluate(stream=stream, model=[hy, hat],
              model_names=['HT', 'HAT']);
```

The summary of the evaluation is:

```
Processed samples: 20000
Mean performance:
HT - Accuracy     : 0.7279
HT - Kappa        : 0.4530
HT - Size (kB)          : 175.8711
HAT - Accuracy     : 0.8070
HAT - Kappa        : 0.6122
HAT - Size (kB)         : 122.0986
```

The result of this experiment is shown in Fig. 5. During the first 5K samples, we see that both methods behave in a very similar way, which is expected as the `HoeffdingAdaptiveTreeClassifier` essentially works as the `HoeffdingTreeClassifier` when there is no drift. At the 5K mark, the first drift is observable by the sudden drop in the performance of both estimators. However, notice that the `HoeffdingAdaptiveTreeClassifier` has the edge and recovers faster. The same behavior is observed after the drift in the 15K mark. Interestingly, after the drift at 10K,

the `HoeffdingTreeClassifier` is better for a small period but is quickly overtaken. In this experiment, we can also see that the *current* performance evaluation provides richer insights on the performance of each estimator. It is worth noting the difference in memory between these estimators. The `HoeffdingAdaptiveTreeClassifier` achieves better performance while requiring less space in memory. This indicates that the branch replacement mechanism triggered by ADWIN has been applied, resulting in a less complex tree structure representing the data.

Real-time applications

We recognize that previous examples use static synthetic data for illustrative purposes. However, the goal is to work on real-world streaming applications where data is continuously generated and must be processed in real-time. In this context, scikit-multiflow is designed to interact with specialized streaming tools, providing flexibility to the users to deploy streaming models and tools in different environments. For instance, an IoT architecture on an edge/fog/cloud computing environment is proposed in [CW19]. This architecture is designed to capture, manage, process, analyze, and visualize IoT data streams. In this architecture, scikit-multiflow is the stream machine learning library inside the processing and analytics block.

In the following example, we show how we can leverage existing Python tools to interact with dynamic data. We use *Streamz*[8] to get data from Apache Kafka. The data from the stream is used to incrementally train, one sample at a time, a `HoeffdingTreeClassifier` model. The output on each iteration is a boolean value indicating if the model correctly classified the last sample from the stream.

```
from streamz import Stream
from skmultiflow.trees import HoeffdingTreeClassifier

@Stream.register_api()
class extended(Stream):
    def __init__(self, upstream, model, **kwargs):
        self.model = model
        super().__init__(upstream, **kwargs)

    def update(self, x, who=None):
        # Tuple x represents one data sample
        # x[0] is the features array and
        # x[1] is the target label
        y_pred = self.model.predict(x[0])
        # incrementally learn the current sample
        self.model.partial_fit(x[0], x[1])
        # output indicating if the model
        # correctly classified the sample
        self._emit(y_pred == x[1])

s_in = Stream.from_kafka(**config)
ht = HoeffdingTreeClassifier()

s_learn = s.map(read).extended(model=ht)
out = s_learn.sink_to_list()

s_in.start()
```

Alternatively, we could define two nodes, one for training and one for predicting. In this case, we just need to make sure that we maintain the *test-then-train* order.

8. https://github.com/python-streamz/streamz

Get scikit-multiflow

scikit-multiflow work with Python 3.5+ and can be used on Linux, macOS, and Windows. The source code is publicly available in GitHub. The stable release version is available via `conda-forge` (recommended) and `pip`:

```
$ conda install -c conda-forge scikit-multiflow

$ pip install -U scikit-multiflow
```

The latest development version is available in the project's repository: https://github.com/scikit-multiflow/scikit-multiflow. Stable and development versions are also available as `docker` images.

Conclusions and final remarks

In this paper, we provide a brief overview of machine learning for data streams. Stream learning is an alternative to standard batch learning in dynamic environments where data is continuously generated (potentially infinite) and data is non-stationary but evolves (concept drift). We present examples of applications and describe the challenges and requirements of machine learning techniques to be used on streaming data effectively and efficiently.

We describe scikit-multiflow, an open-source machine learning library for data streams in Python. The design of scikit-multiflow is based on two principles: to be easy to design and run experiments, and to be easy to extend and modify existing methods. We provide a quick overview of the core elements of scikit-multiflow and show how it can be used for the tasks of classification and drift detection.

Acknowledgments

The author is particularly grateful to Prof. Albert Bifet from the Department of Computer Science at the University of Waikato for his continuous support. We also thank Saulo Martiello Mastelini from the Institute of Mathematics and Computer Sciences at the University of São Paulo, for his active collaboration on scikit-multiflow and his valuable work as one of the maintainers of the project. We thank interns who have helped in the development of scikit-multiflow and the open-source community which motivates and contributes in the continuous improvement of this project. We gratefully acknowledge the reviewers from the SciPy 2020 conference for their constructive comments.

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.

[BG07] Albert Bifet and Ricard Gavalda. Learning from Time-Changing Data with Adaptive Windowing. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 443–448, 2007. doi:10.1137/1.9781611972771.42.

[BG09] Albert Bifet and Ricard Gavalda. Adaptive Learning from Evolving Data Streams. In *8th International Symposium on Intelligent Data Analysis*, pages 249–260, 2009. doi:10.1007/978-3-642-03915-7_22.

[BHKP11] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Data stream mining a practical approach, 2011.

[Coh60] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960. doi:10.1177/001316446002000104.

[CW19] Hung Cao and Monica Wachowicz. Analytics everywhere for streaming iot data. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 18–25, 2019. doi:10.1109/IOTSMS48152.2019.8939171.

[Daw84] A Philip Dawid. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, pages 278–292, 1984.

[DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80. ACM, 2000.

[GRB+19] Heitor Murilo Gomes, Jesse Read, Albert Bifet, Jean Paul Barddal, and João Gama. Machine learning for streaming data: State of the art, challenges, and opportunities. *SIGKDD Explor. Newsl.*, 21(2):6–22, 2019. doi:10.1145/3373464.3373470.

[GZB+14] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37, 2014. doi:10.1145/2523813.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume 14, pages 1137–1145, 1995.

[MRBA18] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. Scikit-Multiflow: A Multi-output Streaming Framework. *Journal of Machine Learning Research*, 19(72):1–5, 2018. URL: http://jmlr.org/papers/v19/18-251.html.

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

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[vCV11] 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, 2011. doi:10.1109/MCSE.2011.37.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:https://doi.org/10.1038/s41592-019-0686-2.

Awkward Array: JSON-like data, NumPy-like idioms

Jim Pivarski[‡*], Ianna Osborne[‡], Pratyush Das[¶], Anish Biswas[§], Peter Elmer[‡]

https://youtu.be/WlnUF3LRBj4

Abstract—NumPy simplifies and accelerates mathematical calculations in Python, but only for rectilinear arrays of numbers. Awkward Array provides a similar interface for JSON-like data: slicing, masking, broadcasting, and performing vectorized math on the attributes of objects, unequal-length nested lists (i.e. ragged/jagged arrays), and heterogeneous data types.

Awkward Arrays are columnar data structures, like (and convertible to/from) Apache Arrow, with a focus on manipulation, rather than serialization/transport. These arrays can be passed between C++ and Python, and they can be used in functions that are JIT-compiled by Numba.

Development of a GPU backend is in progress, which would allow data analyses written in array-programming style to run on GPUs without modification.

Index Terms—NumPy, Numba, Pandas, C++, Apache Arrow, Columnar data, AOS-to-SOA, Ragged array, Jagged array, JSON

Introduction

NumPy [np] is a powerful tool for data processing, at the center of a large ecosystem of scientific software. Its built-in functions are general enough for many scientific domains, particularly those that analyze time series, images, or voxel grids. However, it is difficult to apply NumPy to tasks that require data structures beyond N-dimensional arrays of numbers.

More general data structures can be expressed as JSON and processed in pure Python, but at the expense of performance and often conciseness. NumPy is faster and more memory efficient than pure Python because its routines are precompiled and its arrays of numbers are packed in a regular way in contiguous memory. Some expressions are more concise in NumPy's "vectorized" notation, which describe actions to perform on whole arrays, rather than scalar values.

In this paper, we describe Awkward Array [ak1], [ak2], a generalization of NumPy's core functions to the nested records, variable-length lists, missing values, and heterogeneity of JSON-like data. The internal representation of these data structures is columnar, very similar to (and compatible with) Apache Arrow [arrow]. But unlike Arrow, the focus of Awkward Array is to provide a suite of data manipulation routines, just as NumPy's role is focused on transforming arrays, rather than standardizing a serialization format.

* *Corresponding author: pivarski@princeton.edu*
‡ *Princeton University*
¶ *Institute of Engineering and Management*
§ *Manipal Institute of Technology*

Our goal in developing Awkward Array is not to replace NumPy, but to extend the set of problems to which it can be applied. We use NumPy's extension mechanisms to generalize its interface in a way that returns identical output where the applicability of the two libraries overlap (i.e. rectilinear arrays), and the implementation of non-structure-changing, numerical math is deferred to NumPy itself. Thus, all the universal functions (ufuncs) in the SciPy project [scipy] and its ecosystem can already be applied to Awkward structures because they inherit NumPy and SciPy's own implementations.

Origin and development

Awkward Array was intended as a way to enable particle physics analyses to take advantage of scientific Python tools. Particle physics problems are inherently structured, frequently needing nested loops over variable-length lists. They also involve big data, typically tens to hundreds of terabytes per analysis. Traditionally, this required physicists to do data analysis in Fortran (with custom libraries for data structures [hydra] before Fortran 90) and C++, but many physicists are now moving to Python for end-stage analysis [phypy]. Awkward Array provides the link between scalable, interactive, NumPy-based tools and the nested, variable-length data structures that physicists need.

Since its release in September 2018, Awkward Array has become one of the most popular Python libraries for particle physics, as shown in Figure 1. The Awkward 0.x branch was written using NumPy only, which limited its development because every operation must be vectorized for performance. We (the developers) also made some mistakes in interface design and learned from the physicists' feedback.

Spurred by these shortcomings and the popularity of the general concept, we redesigned the library as Awkward 1.x in a half-year project starting in August 2019. The new library is compiled as an extension module to allow us to write custom precompiled loops, and its Python interface is improved: it is now a strict generalization of NumPy, is compatible with Pandas [pandas] (Awkward Arrays can be DataFrame columns), and is implemented as a Numba [numba] extension (Awkward Arrays can be used in Numba's just-in-time compiled functions).

Although the Awkward 1.x branch is feature-complete, serialization to and from a popular physics file format (ROOT [root], which represents over an exabyte of physics data [root-EB]) is not. Adoption among physicists is ongoing, but the usefulness of JSON-like structures in data analysis is not domain-specific and should be made known to the broader community.

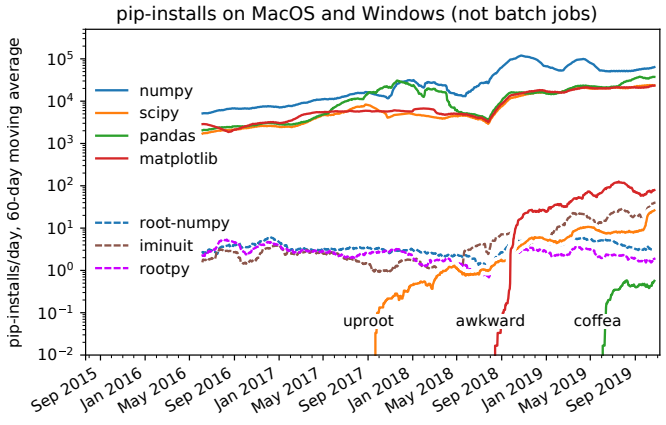


Fig. 1: *Adoption of Awkward 0.x, measured by PyPI statistics, compared to other popular particle physics packages (root-numpy, iminuit, rootpy) and popular data science packages.*

Demonstration using a GeoJSON dataset

To show how Awkward Arrays can be applied beyond particle physics, this section presents a short exploratory analysis of Chicago bike routes [bikes] in GeoJSON format. GeoJSON has a complex structure with multiple levels of nested records and variable-length arrays of numbers, as well as strings and missing data. These structures could not be represented as a NumPy array (without `dtype=object`, which are Python objects wrapped in an array), but there are reasons to want to perform NumPy-like math on the numerical longitude, latitude coordinates.

To begin, we load the publicly available GeoJSON file,

```
import urllib.request
import json

url = "https://raw.githubusercontent.com/Chicago/" \
      "osd-bike-routes/master/data/Bikeroutes.geojson"
bikeroutes_json = urllib.request.urlopen(url).read()
bikeroutes_pyobj = json.loads(bikeroutes_json)
```

and convert it to an Awkward Array. The two main data types are `ak.Array` (a sequence of items, which may contain records) and `ak.Record` (a single object with named, typed fields, which may contain arrays). Since the dataset is a single JSON object, we pass it to the `ak.Record` constructor.

```
import awkward1 as ak
bikeroutes = ak.Record(bikeroutes_pyobj)
```

The record-oriented structure of the JSON object, in which fields of the same object are serialized next to each other, has now been transformed into a columnar structure, in which data from a single field across all objects are contiguous in memory. This requires more than one buffer in memory, as heterogeneous data must be split into separate buffers by type.

The structure of this particular file (expressed as a Datashape, obtained by calling `ak.type(bikeroutes)`) is

```
{"type": string,
 "crs": {
     "type": string,
     "properties": {"name": string}},
 "features": var * {
     "type": string,
     "properties": {
         "STREET": string,
         "TYPE": string,
         "BIKEROUTE": string,
         "F_STREET": string,
         "T_STREET": option[string]},
     "geometry": {
         "type": string,
         "coordinates":
             var * var * var * float64}}}
```

We are interested in the longitude, latitude coordinates, which are in the `"coordinates"` field of the `"geometry"` of the `"features"`, at the end of several levels of variable-length lists (`var`). At the deepest level, longitude values are in coordinate `0` and latitude values are in coordinate `1`.

We can access each of these, eliminating all other fields, with a NumPy-like multidimensional slice. Strings in the slice select fields of records and ellipsis (`...`) skips dimensions as it does in NumPy.

```
longitude = bikeroutes["features", "geometry",
                       "coordinates", ..., 0]
latitude  = bikeroutes["features", "geometry",
                       "coordinates", ..., 1]
```

The `longitude` and `latitude` arrays both have type `1061 * var * var * float64`; that is, 1061 routes with a variable number of variable-length polylines.

At this point, we might want to compute the length of each route, and we can use NumPy ufuncs to do that, despite the irregular shape of the `longitude` and `latitude` arrays. First, we subtract off the mean and convert degrees into a unit of distance (`82.7` and `111.1` are conversion factors at Chicago's latitude).

```
km_east = (longitude - np.mean(longitude)) * 82.7
km_north = (latitude - np.mean(latitude)) * 111.1
```

Subtraction and multiplication defer to `np.subtract` and `np.multiply`, respectively, and these are ufuncs, overridden using NumPy's `__array_ufunc__` protocol [nep13]. The `np.mean` function is not a ufunc, but it, too, can be overridden using the `__array_function__` protocol [nep18]. All ufuncs and a handful of more generic functions can be applied to Awkward Arrays.

To compute distances between points in an array `a` in NumPy, we would use an expression like the following,

```
differences = a[1:] - a[:-1]
```

which views the same array without the first element (`a[1:]`) and without the last element (`a[:-1]`) to subtract "between the fenceposts." We can do so in the nested lists with

```
differences = km_east[:, :, 1:] - km_east[:, :, :-1]
```

even though the first two dimensions have variable lengths. They're derived from the same array (`km_east`), so they have the same lengths and every element in the first term can be paired with an element in the second term.

Two-dimensional distances are the square root of the sum of squares of these differences,

```
segment_length = np.sqrt(
    (km_east[:, :, 1:] - km_east[:, :, :-1])**2 +
    (km_north[:, :, 1:] - km_north[:, :, :-1])**2)
```

and we can sum up the lengths of each segment in each polyline in each route by calling `np.sum` on the deepest `axis`.

```
polyline_length = np.sum(segment_length, axis=-1)
route_length = np.sum(polyline_length, axis=-1)
```

The same could be performed with the following pure Python code, though the vectorized form is shorter, more exploratory, and 8× faster (Intel 2.6 GHz i7-9750H processor with 12 MB cache on a single thread); see Figure 2.

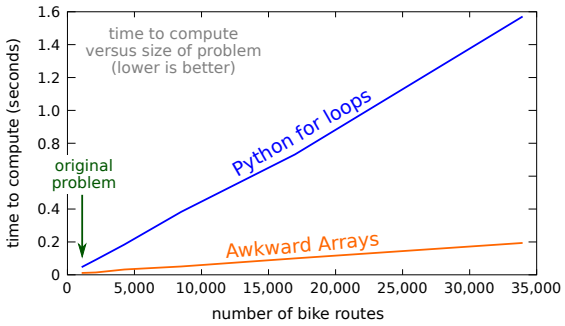


Fig. 2: *Scaling of Awkward Arrays and pure Python loops for the bike routes calculation shown in the text.*

```
route_length = []
for route in bikeroutes_pyobj["features"]:
    polyline_length = []
    for polyline in route["geometry"]["coordinates"]:
        segment_length = []
        last = None
        for lng, lat in polyline:
            km_east = lng * 82.7
            km_north = lat * 111.1
            if last is not None:
                dx2 = (km_east - last[0])**2
                dy2 = (km_north - last[1])**2
                segment_length.append(
                    np.sqrt(dx2 + dy2))
            last = (km_east, km_north)

        polyline_length.append(sum(segment_length))
    route_length.append(sum(polyline_length))
```

The performance advantage is due to Awkward Array's precompiled loops, though this is mitigated by the creation of intermediate arrays and many passes over the same data (once per user-visible operation). When the single-pass Python code is just-in-time compiled by Numba *and* evaluated over Awkward Arrays, the runtime is 250× faster than pure Python (same architecture).

Scope: data types and common operations

Awkward Array supports the same suite of abstract data types and features as "typed JSON" serialization formats—Arrow, Parquet, Protobuf, Thrift, Avro, etc. Namely, there are

- primitive types: numbers and booleans,
- variable-length lists,
- regular-length lists as a distinct type (i.e. tensors),
- records/structs/objects (named, typed fields),
- fixed-width tuples (unnamed, typed fields),
- missing/nullable data,
- mixed, yet specified, types (i.e. union/sum types),
- virtual arrays (functions generate arrays on demand),
- partitioned arrays (for off-core and parallel analysis).

Like Arrow and Parquet, arrays with these features are laid out as columns in memory (more on that below).

Like NumPy, the Awkward Array library consists of a primary Python class, `ak.Array`, and a collection of generic operations. Most of these operations change the structure of the data in the array, since NumPy, SciPy, and others already provide numerical math as ufuncs.

Awkward functions include

- basic and advanced slices (`__getitem__`) including variable-length and missing data as advanced slices,
- masking, an alternative to slices that maintains length but introduces missing values instead of dropping elements,
- broadcasting of universal functions into structures,
- reducers of and across variable-length lists,
- zip/unzip/projecting free arrays into and out of records,
- flattening and padding to make rectilinear data,
- Cartesian products (cross join) and combinations (self join) at `axis >= 1` (per element of one or more arrays).

Conversions to other formats, such as Arrow, and interoperability with common Python libraries, such as Pandas and Numba, are also in the library's scope.

Columnar representation, columnar implementation

Awkward Arrays are columnar, not record-oriented, data structures. Instead of concentrating all data for one array element in nearby memory (as an "array of structs"), all data for a given field are contiguous, and all data for another field are elsewhere contiguous (as a "struct of arrays"). This favors a pattern of data access in which only a few fields are needed at a time, such as the longitude, latitude coordinates in the bike routes example.

Additionally, Awkward operations are performed on columnar data without returning to the record-oriented format. To illustrate, consider an array of variable-length lists, such as the following toy example:

```
[[1.1, 2.2, 3.3], [4.4], [5.5, 6.6], [7.7, 8.8, 9.9]]
```

Instead of creating four C++ objects to represent the four lists, we can put all of the numerical data in one buffer and indicate where the lists start and stop with two integer arrays:

```
starts:  0, 3, 4, 6
stops:   3, 4, 6, 9
content: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9
```

For an array of lists of lists, we could introduce two levels of `starts` and `stops` arrays, one to specify where the outer square brackets start and stop, another to specify the inner square brackets. Any tree-like data structure can be built in this way; the hierarchy of nested array groups mirrors the hierarchy of the nested data, except that the number of these nodes scales with the complexity of the data type, not the number of elements in the array. Particle physics use-cases require thousands of nodes to describe complex collision events, but billions of events in memory at a time. Figure 3 shows a small example.

In the bike routes example, we computed distances using slices like `km_east[:, :, 1:]`, which dropped the first element from each list. In the implementation, list objects are not created for the sake of removing one element before translating back into a columnar format; the operation is performed directly on the columnar data.

For instance, to drop the first element from each list in an array of lists `a`, we only need to add `1` to the `starts`:

```
starts:  1, 4, 5, 7
stops:   3, 4, 6, 9
content: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9
```

Without modifying the `content`, this new array represents

```
[[     2.2, 3.3], [   ], [     6.6], [     8.8, 9.9]]
```

because the first list starts at index `1` and stops at `3`, the second starts at `4` and ends at `4`, etc. The "removed" elements are still present in the `content` array, but they are now unreachable, much like the stride tricks used for slicing in NumPy.

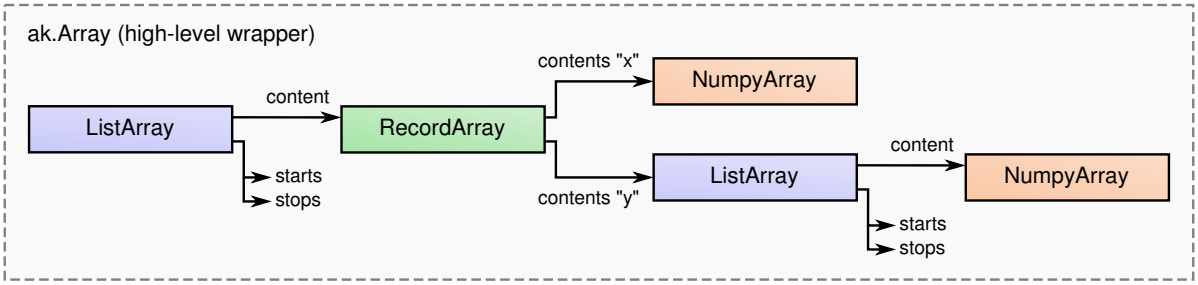


Fig. 3: *Hierarchy for an example data structure: an array of lists of records, in which field* `"x"` *of the records are numbers and field* `"y"` *of the records are lists of numbers. This might, for example, represent* `[[], [{"x": 1, "y": [1]}, {"x": 2, "y": [2, 2]}]]`*, but it also might represent an array with billions of elements (of the same type). The number of nodes scales with complexity, not data volume.*

Leaving the `content` untouched means that the precompiled slice operation does not depend on the `content` type, not even whether the `content` is a numeric array or a tree structure, as in Figure 3. It also means that this operation does not cascade down such a tree structure, if it exists. Most operations leave nested structure untouched and return views, rather than copies, of most of the input buffers.

Architecture of Awkward 1.x

In August 2019, we began a half-year project to rewrite the library in C++ (Awkward 1.x), which is now complete. Whereas Awkward 0.x consists of Python classes that call NumPy on internal arrays to produce effects like the slice operation described in the previous section, Awkward 1.x consists of C++ classes that perform loops in custom compiled code, wrapped in a Python interface through pybind11.

However, the distinction between slow, bookkeeping code and fast math enforced by Python and NumPy is a useful one: we maintained that distinction by building Awkward 1.x in layers that separate the (relatively slow) polymorphic C++ classes, whose job is to organize and track the ownership of data buffers, from the optimized loops in C that manipulate data in those buffers.

These layers are fully broken down below and in Figure 4:

- The high-level interface is in Python.
- The array nodes (managing node hierarchy and ownership/lifetime) are in C++, accessed through pybind11.
- An alternate implementation of array navigation was written for Python functions that are compiled by Numba.
- Array manipulation algorithms (without memory management) are independently implemented as "cpu-kernels" and "cuda-kernels" plugins. The kernels' interface is pure C, allowing for reuse in other languages.

The separation of "kernels" from "navigation" has two advantages: (1) optimization efforts can focus on the kernels, since these are the only loops that scale with data volume, and (2) CPU-based kernels can, in principle, be swapped for GPU-based kernels. The latter is an ongoing project.

Numba for just-in-time compilation

Some expressions are simpler in "vectorized" form, such as the Awkward Array solution to the bike routes calculation. Others are simpler to express as imperative code. This issue arose repeatedly as physicists used Awkward Array 0.x in real problems, both

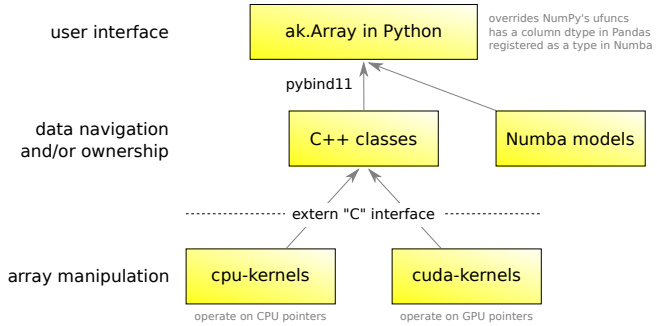


Fig. 4: *Components of Awkward Array, as described in the text. All components have been implemented except for the "cuda-kernels."*

because they were more familiar with imperative code (in C++) and because the problems truly favored non-vectorized solutions. For instance, walking up a tree, looking for nodes of a particular type (such as a tree of particle decays) is hard to express in vectorized form because some elements of a test array reach the stopping condition before others; preventing them from continuing to walk the tree adds complexity to a data analysis. Any problem that must "iterate until converged" is also of this form.

These problems are readily solved by Numba, a just-in-time compiler for Python, but Numba cannot compile code involving arrays from Awkward 0.x. To solve physics problems, we had to break the array abstraction described above. Ensuring that Numba would recognize Awkward 1.x arrays was therefore a high priority, and it is a major component of the final system.

Numba has an extension mechanism for registering new types and overloading operators for new types. We added Numba extensions for the `ak.Array` and `ak.Record` types, overloading `__getitem__` (square bracket) and `__getattr__` (dot) operators and iterators, so that users can walk over the data structures with conventional loops.

Returning to the bike routes example, the following performs the same calculation with Numba:

```
import numba as nb

@nb.jit
def compute_lengths(bikeroutes):
    # allocate output array
    route_length = np.zeros(len(bikeroutes["features"]))

    # loop over routes
    for i in range(len(bikeroutes["features"])):
        route = bikeroutes["features"][i]
```

```
        # loop over polylines
        for polyline in route["geometry"]["coordinates"]:
            first = True
            last_east = 0.0
            last_north = 0.0

            for lng_lat in polyline:
                km_east = lng_lat[0] * 82.7
                km_north = lng_lat[1] * 111.1

                # compute distances between points
                if not first:
                    dx2 = (km_east - last_east)**2
                    dy2 = (km_north - last_north)**2
                    distance = np.sqrt(dx2 + dy2)
                    route_length[i] += distance

                # keep track of previous value
                first = False
                last_east = km_east
                last_north = km_north

    return route_length
```

This expression is not concise, but it is 250× faster than the pure Python solution and 30× faster than even the Awkward Array (precompiled) solution. It makes a single pass over all buffers, maximizing CPU cache efficiency, and it does not allocate or fill any intermediate arrays. This is possible because `nb.jit` compiles specialized machine code for this particular problem.

Combining Awkward Array with Numba has benefits that neither has alone. Ordinarily, complex data structures would have to be passed into Numba as Python objects, which means a second copy of the data that must be "unboxed" (converted into a compiler-friendly form) and "boxed" (converted back). If the datasets are large, this consumes memory and time. Awkward Arrays use less memory than the equivalent Python objects (5.2× smaller for the bike routes) and they use the same internal representation (columnar arrays) inside and outside functions just-in-time compiled by Numba.

The disadvantage of Numba and Awkward Arrays in Numba is that neither support the whole language: Numba can only compile a subset of Python and the NumPy library and Awkward Arrays are limited to imperative-style access (no array-at-a-time functions) and homogeneous data (no union type). Any code that works in a just-in-time compiled function works without compilation, but not vice-versa. Thus, there is a user cost to preparing a function for compilation, which can be seen in a comparison of the code listing above with the pure Python example in the original bike routes section. However, this finagling is considerably less time-consuming than translating a Python function to a language like C or C++ and converting the data structures. It favors gradual transition of an analysis from no just-in-time compilation to a judicious use of it in the parts of the workflow where performance is critical.

ArrayBuilder: creating columnar data in-place

Awkward Arrays are immutable; NumPy's ability to assign elements in place is not supported or generalized by the Awkward Array library. (As an exception, users can assign fields to records using `__setitem__` syntax, but this *replaces* the inner tree with one having the new field.) Restricting Awkward Arrays to read-only access allows whole subtrees of nodes to be shared among different versions of an array.

To create new arrays, we introduced `ak.ArrayBuilder`, an append-only structure that accumulates data and creates `ak.Arrays` by taking a "snapshot" of the current state. The `ak.ArrayBuilder` is also implemented for Numba, so just-in-time compiled Python can build arbitrary data structures.

The `ak.ArrayBuilder` is a dynamically typed object, inferring its type from the types and order of data appended to it. As elements are added, the `ak.ArrayBuilder` builds a tree of columns *and* their types to refine the inferred type.

```
                    # type of b.snapshot()
b                   # 0 * unknown
b.begin_record()    # 0 * {}
b.field("x")        # 0 * {"x": unknown}
b.integer(1)        # 0 * {"x": int64}
b.end_record()      # 1 * {"x": int64}
b.begin_record()    # 1 * {"x": int64}
b.field("x")        # 1 * {"x": int64}
b.real(2.2)         # 1 * {"x": float64}
b.field("y")        # 1 * {"x": float64, "y": ?unknown}
b.integer(2)        # 1 * {"x": float64, "y": ?int64}
b.end_record()      # 2 * {"x": float64, "y": ?int64}
b.null()            # 3 * ?{"x": float64, "y": ?int64}
b.string("hello")   # 4 * ?union[{"x": float64,
                    #             "y": ?int64}, string]
```

In the above example, an initially empty `ak.ArrayBuilder` named `b` has unknown type and zero length. With `begin_record`, its type becomes a record with no fields. Calling `field` adds a field of unknown type, and following that with `integer` sets the field type to an integer. The length of the array is only increased when the record is closed by `end_record`.

In the next record, field `"x"` is filled with a floating point number, which retroactively updates previous integers to floats. Calling `b.field("y")` introduces a field `"y"` to all records, though it has option type because this field is missing for all previous records. The third record is missing (`b.null()`), which refines its type as optional, and in place of a fourth record, we append a string, so the type becomes a union.

Internally, `ak.ArrayBuilder` maintains a similar tree of array buffers as an `ak.Array`, except that all buffers can grow (when the preallocated space is used up, the buffer is reallocated and copied into a buffer 1.5× larger), and `content` nodes can be replaced from specialized types to more general types. Taking a snapshot *shares* buffers with the new array, so it is a lightweight operation.

Although `ak.ArrayBuilder` is compiled code and calls into it are specialized by Numba, its dynamic typing has a runtime cost: filling NumPy arrays is faster. `ak.ArrayBuilder` trades runtime performance for convenience; faster array-building methods would have to be specialized by type.

High-level behaviors

One of the surprisingly popular uses of Awkward 0.x has been to add domain-specific methods to records and arrays by subclassing their hierarchical node types. These can act on scalar records returning scalars, like a C++ or Python object,

```
# distance between points1[0] and points2[0]
points1[0].distance(points2[0])
```

or they may be "vectorized," like a ufunc,

```
# distance between all points1[i] and points2[i]
points1.distance(points2)
```

This capability has been ported to Awkward 1.x and expanded upon. In Awkward 1.x, records can be named (as part of more

general "properties" metadata in C++) and record names are linked to Python classes through an `ak.behavior` dict.

```
class Point:
    def distance(self, other):
        return np.sqrt((self.x - other.x)**2 +
                       (self.y - other.y)**2)

class PointRecord(Point, ak.Record):
    pass

class PointArray(Point, ak.Array):
    pass

ak.behavior["point"] = PointRecord
ak.behavior["*", "point"] = PointArray

points1 = ak.Array([{"x": 1.1, "y": 1},
                    {"x": 2.2, "y": 2},
                    {"x": 3.3, "y": 3}],
                   with_name="point")

points2 = ak.Array([{"x": 1, "y": 1.1},
                    {"x": 2, "y": 2.2},
                    {"x": 3, "y": 3.3}],
                   with_name="point")

points1[0].distance(points2[0])
# 0.14142135623730964

points1.distance(points2)
# <Array [0.141, 0.283, 0.424] type='3 * float64'>

points1.distance(points2[0])   # broadcasting
<Array [0.141, 1.5, 2.98] type='3 * float64'>
```

When an operation on array nodes completes and the result is wrapped in a high-level `ak.Array` or `ak.Record` class for the user, the `ak.behavior` is checked for signatures that link records and arrays of records to user-defined subclasses. Only the name `"point"` is stored with the data; methods are all added at runtime, which allows schemas to evolve.

Other kinds of behaviors can be assigned through different signatures in the `ak.behavior` dict, such as overriding ufuncs,

```
# link np.absolute("point") to a custom function
def magnitude(point):
    return np.sqrt(point.x**2 + point.y**2)

ak.behavior[np.absolute, "point"] = magnitude

np.absolute(points1)
# <Array [1.49, 2.97, 4.46] type='3 * float64'>
```

as well as custom broadcasting rules, and Numba extensions (typing and lowering functions).

As a special case, strings are not defined as an array type, but as a parameter label on variable-length lists. Behaviors that present these lists as strings (overriding `__repr__`) and define per-string equality (overriding `np.equal`) are preloaded in the default `ak.behavior`.

Awkward Arrays and Pandas

Awkward Arrays are registered as a Pandas extension, so they can be losslessly embedded within a `Series` or a `DataFrame` as a column. Some Pandas operations can be performed on them—particularly, NumPy ufuncs and any high-level behaviors that override ufuncs—but best practices for using Awkward Arrays within Pandas are largely unexplored. Most Pandas functions were written without deeply nested structures in mind.

It is also possible (and perhaps more useful) to translate Awkward Arrays into Pandas's own ways of representing nested structures. Pandas's MultiIndex is particularly useful: variable-length lists translate naturally into MultiIndex rows:

```
ak.pandas.df(ak.Array([[[1.1, 2.2], [], [3.3]],
                       [],
                       [[4.4], [5.5, 6.6]],
                       [[7.7]],
                       [[8.8]]]))
#                                values
# entry subentry subsubentry
# 0     0        0               1.1
#                1               2.2
#        2        0               3.3
# 2     0        0               4.4
#        1        0               5.5
#                 1               6.6
# 3     0        0               7.7
# 4     0        0               8.8
```

and nested records translate into MultiIndex column names:

```
ak.pandas.df(ak.Array([{"I":
                          {"a": _, "b": {"c": _}},
                        "II":
                          {"x": {"y": {"z": _}}}}
                       for _ in range(0, 50, 10)]))
#          I       II
#          a   b    x
#              c    y
#                   z
# entry
# 0        0   0    0
# 1       10  10   10
# 2       20  20   20
# 3       30  30   30
# 4       40  40   40
```

In the first of these two examples, empty lists in the Awkward Array do not appear in the Pandas output, though their existence may be inferred from gaps between entry and subentry indexes. When analyzing both lists and non-list data, or lists of different lengths, it is more convenient to translate an Awkward Array into multiple DataFrames and `JOIN` those DataFrames as relational data than to try to express it all in one DataFrame.

This example highlights a difference in applicability between Pandas and Awkward Array: Pandas is better at solving problems with long-range relationships, joining on relational keys, but the structures that a single DataFrame can represent (without resorting to Python objects) is limited. Awkward Array allows general data structures with different length lists in the same structure, but most calculations are elementwise, as in NumPy.

GPU backend

One of the advantages of a vectorized user interface is that it is already optimal for calculations on a GPU. Imperative loops have to be redesigned when porting algorithms to GPUs, but CuPy, Torch, TensorFlow, and JAX demonstrate that an interface consisting of array-at-a-time functions hides the distinction between CPU calculations and GPU calculations, making the hardware transparent to users.

Partly for the sake of adding a GPU backend, all instances of reading or writing to an array's buffers were restricted to the "array manipulation" layer of the project (see Figure 4). The first implementation of this layer, "cpu-kernels," performs all operations that actually access the array buffers, and it is compiled into a physically separate file: `libawkward-cpu-kernels.so`, as opposed to the main `libawkward.so`, Python extension module, and Python code.

In May 2020, we began developing the "cuda-kernels" library, provisionally named `libawkward-cuda-kernels.so`

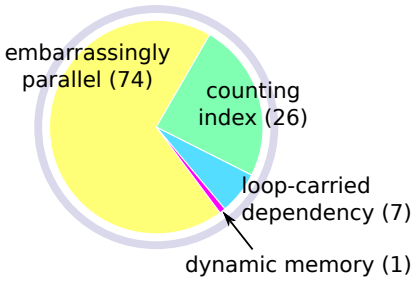


Fig. 5: *CPU kernels by algorithmic complexity, as of February 2020.*

(to allow for future non-CUDA versions). Since the main codebase (`libawkward.so`) never dereferences any pointers to its buffers, main memory pointers can be transparently swapped for GPU pointers with additional metadata to identify which kernel to call for a given set of pointers. Thus, the main library does not need to be recompiled to support GPUs and it can manage arrays in main memory and on GPUs in the same process, which could be important, given the limited size of GPU memory. The "cuda-kernels" will be deployed as a separate package in PyPI and Conda so that users can choose to install it separately as an "extras" package.

The kernels library contains many functions (428 in the `"extern C"` interface with 124 independent implementations, as of May 2020) because it defines all array manipulations. All of these must be ported to CUDA for the first GPU implementation. Fortunately, the majority are easy to translate: Figure 5 shows that almost 70% are simple, embarrassingly parallel loops, 25% use a counting index that could be implemented with a parallel prefix sum, and the remainder have loop-carried dependencies or worse (one used dynamic memory, but it has since been rewritten). The kernels were written in a simple style that may be sufficiently analyzable for machine-translation, a prospect we are currently investigating with pycparser.

Transition from Awkward 0.x

Awkward 0.x is popular among physicists, and some data analyses have come to depend on it and its interface. User feedback, however, has taught us that the Awkward 0.x interface has some inconsistencies, confusing names, and incompatibilities with NumPy that would always be a pain point for beginners if maintained, yet ongoing analyses must be supported. (Data analyses, unlike software stacks, have a finite lifetime and can't be required to "upgrade or perish," especially when a student's graduation is at stake.)

To support both new and ongoing analyses, we gave the Awkward 1.x project a different Python package name and PyPI package name from the original Awkward Array: `awkward1` versus `awkward`. This makes it possible to install both and load both in the same process (unlike Python 2 and Python 3). Conversion functions have also been provided to aid in the transition.

We are already recommending Awkward 1.x for new physics analyses, even though serialization to and from the popular ROOT file format is not yet complete. Nevertheless, the conversion functions introduce an extra step and we don't expect widespread adoption until the Uproot library natively converts ROOT data to and from Awkward 1.x arrays.

Eventually, however, it will be time to give Awkward 1.x "official" status by naming it `awkward` in Python and PyPI. At that time, Awkward 0.x will be renamed `awkward0`, so that a single

```
import awkward0 as awkward
```

would be required to maintain old analysis scripts.

As an incentive for adopting Awkward 1.x in new projects, it has been heavily documented, with complete docstring and doxygen coverage (already exceeding Awkward 0.x).

Summary

By providing NumPy-like idioms on JSON-like data, Awkward Array satisfies a need required by the particle physics community. The inclusion of data structures in array types and operations was an enabling factor in this community's adoption of other scientific Python tools. However, the Awkward Array library itself is not domain-specific and is open to use in other domains. We are very interested in applications and feedback from the wider data analysis community.

Acknowledgements

Support for this work was provided by NSF cooperative agreement OAC-1836650 (IRIS-HEP), grant OAC-1450377 (DIANA/HEP) and PHY-1520942 (US-CMS LHC Ops).

References

[np] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[ak1] Jim Pivarski, Jaydeep Nandi, David Lange, Peter Elmer. *Columnar data processing for HEP analysis*, Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018). DOI:10.1051/epjconf/201921406026

[ak2] Jim Pivarski, Peter Elmer, David Lange. *Awkward Arrays in Python, C++, and Numba*, CHEP 2019 proceedings, EPJ Web of Conferences (CHEP 2019). arxiv:2001.06307

[arrow] Apache Software Foundation. *Arrow: a cross-language development platform for in-memory data*, https://arrow.apache.org

[scipy] Pauli Virtanen et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, in press. DOI:10.1038/s41592-019-0686-2

[hydra] R. K. Böck. *Initiation to Hydra*, https://cds.cern.ch/record/864527 (1974), DOI:10.5170/CERN-1974-023.402

[phypy] Jim Pivarski. *Programming languages and particle physics*, https://events.fnal.gov/colloquium/events/event/pivarski-colloq-2019 (2019).

[pandas] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010), DOI:10.25080/Majora-92bf1922-00a

[numba] Siu Kwan Lam, Antoine Pitrou, Stanley Seibert. *Numba: a LLVM-based Python JIT compiler*, LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 7, 1-6 (2015), DOI:10.1145/2833157.2833162

[root] Rene Brun and Fons Rademakers, *ROOT: an object oriented data analysis framework*, Proceedings AIHENP'96 Workshop, Lausanne, (1996), Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.

[root-EB] Axel Naumann. *ROOT as a framework and analysis tool in run 3 and the HL-LHC era*, https://indico.cern.ch/event/913205/contributions/3840338 (2020).

[bikes] City of Chicago Data Portal, https://data.cityofchicago.org

[nep13] Pauli Virtanen, Nathaniel Smith, Marten van Kerkwijk, Stephan Hoyer. *NEP 13 — A Mechanism for Overriding Ufuncs*, https://numpy.org/neps/nep-0013-ufunc-overrides.html

[nep18] Stephan Hoyer, Matthew Rocklin, Marten van Kerkwijk, Hameer Abbasi, Eric Wieser. *NEP 18 — A dispatch mechanism for NumPy's high level array functions*, https://numpy.org/neps/nep-0018-array-function-protocol.html

High-performance operator evaluations with ease of use: libCEED's Python interface

Valeria Barra[§*], Jed Brown[§], Jeremy Thompson[§], Yohann Dudouit[‡]

Abstract—libCEED is a new lightweight, open-source library for high-performance matrix-free Finite Element computations. libCEED offers a portable interface to high-performance implementations, selectable at runtime, tuned for a variety of current and emerging computational architectures, including CPUs and GPUs. libCEED's interface is purely algebraic, facilitating co-design with vendors and enabling unintrusive integration in new and legacy software. In this work, we present libCEED's newly-available Python interface, which opens up new strategies for parallelism and scaling in high-performance Python, without compromising ease of use.

Index Terms—High-performance Python, performance portability, scalability, parallelism, high-order finite elements

Introduction

Historically, high-order Finite Element Methods (FEM) have seen very limited use for industrial problems because the matrix describing the action of the operator loses sparsity as the order is increased [Ors80], leading to unaffordable solve times and memory requirements [Bro10]. Consequently, most industrial applications have used at most quadratic polynomial bases, for which assembled matrices appear to be a good choice, at least when one seeks to minimize the number of floating point operations (FLOPs) per degree of freedom (DOF); see the right panel of Fig. 1. Nowadays, high-order numerical methods, such as the spectral element method (SEM)—a special case of nodal p-Finite Element Method that can reuse the interpolation nodes for quadrature—are employed (e.g., in scientific computing packages such as MFEM [MFE20] and Nek5000 [Nek20]), especially with applications for which implicit solves are limited to linear constant-coefficient separable equations on (nearly) affine elements, which can be efficiently solved with sum factorization and multigrid [LF05].

In Fig. 1, we analyze and compare the asymptotic costs of applying the action of a finite element matrix using different configurations: assembling the sparse matrix representing the action of the operator (labeled as *assembled*), applying the action without assembly while using a tensor-product decomposition of the basis and metric terms computed on the fly with a compact representation of the linearization stored at quadrature points (labeled as *tensor*), and similarly, but with a precomputed pull-back of the linearization to reference elements (labeled as *tensor-qstore*). In the right panel, we show the cost in terms of FLOPs/DOF. This metric for computational efficiency made sense historically, when performance was primarily limited by floating point arithmetic. Memory bandwidth is the overwhelming bottleneck on today's machines, which can perform 40-100 FLOPs per floating point load from memory, and thus the left panel of Fig. 1 becomes a more accurate performance model for modern architectures. We can see that well-implemented high-order methods require low memory motion that decreases with polynomial order and FLOPs that are relatively insensitive to polynomial order for operator evaluation. Thus, high-order methods in matrix-free representation not only possess favorable properties, such as higher accuracy and faster convergence to solution, but also manifest an efficiency gain compared to their corresponding assembled representations.

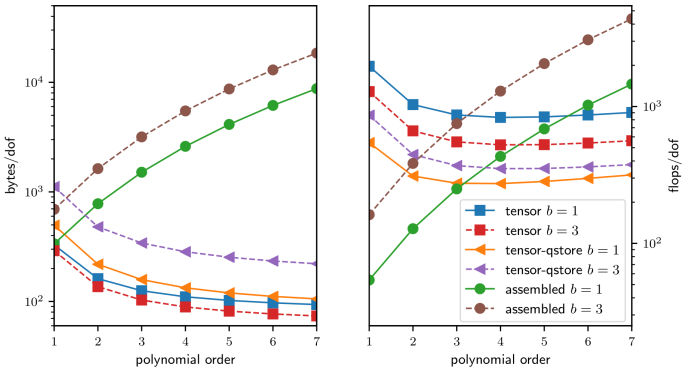


Fig. 1: *Comparison of asymptotic memory transfer and floating point operations per degree of freedom for different representations of a linear operator for a PDE (on a 3D hexahedral mesh) with b components and variable coefficients arising due to Newton linearization of a material nonlinearity. The representation labeled as* tensor *computes metric terms on the fly and stores a compact representation of the linearization at quadrature points. The representation labeled as* tensor-qstore *pulls the metric terms into the stored representation. The* assembled *representation uses a (block) CSR format.*

For the reasons mentioned above, in recent years, high-order numerical methods have been widely used in Partial Differential Equation (PDE) solvers, but software packages that provide high-performance implementations have often been special-purpose and intrusive. In contrast, libCEED [lib20b], the Code for Efficient Extensible Discretizations is light-weight, minimally intrusive, and very versatile. In fact, libCEED offers a purely algebraic interface for matrix-free operator representation and supports run-

* *Corresponding author: valeria.barra@colorado.edu*
§ *University of Colorado Boulder*
‡ *Lawrence Livermore National Laboratory*

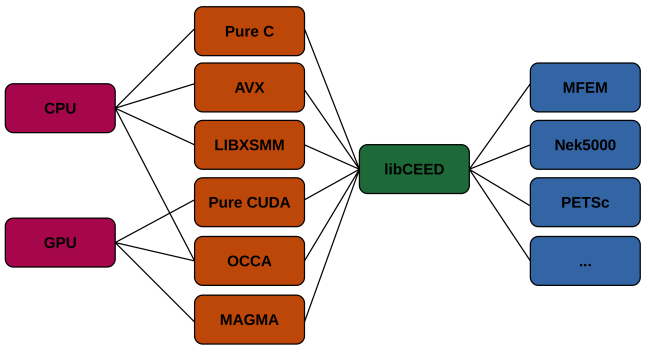


Fig. 2: *The role of libCEED as a lightweight, portable library that provides a low-level API for efficient, specialized implementations. libCEED allows different applications to share highly optimized discretization kernels.*

time selection of implementations tuned for a variety of computational device types, including CPUs and GPUs. libCEED's algebraic interface can unobtrusively be integrated in new and legacy software to provide performance portable interfaces. While libCEED's focus is on high-order finite elements, the approach is algebraic and thus applicable to other discretizations in factored form (e.g., spectral difference). libCEED's role, as a low-level library that allows a wide variety of applications to share highly optimized discretization kernels, is illustrated in Fig. 2, where a non-exhaustive list of specialized implementations (backends) is provided. libCEED provides a low-level Application Programming Interface (API) for user codes so that applications with their own discretization infrastructure (e.g., those in PETSc [BAA+20], MFEM and Nek5000) can evaluate and use the core operations provided by libCEED. GPU implementations are available via pure CUDA [CUD20] as well as the OCCA [OCC20] and MAGMA [MAG20] libraries. CPU implementations are available via pure C and AVX intrinsics as well as the LIBXSMM [LIB20c] library. libCEED provides a unified interface, so that users only need to write a single source code and can select the desired specialized implementation at run time. Moreover, each process or thread can instantiate an arbitrary number of backends.

In this work, we first introduce libCEED's conceptual model and interface, then illustrate its new Python interface, which was developed using the C Foreign Function Interface (CFFI) for Python. CFFI allows reuse of most of the C declarations and requires only a minimal adaptation of some of them. The C and Python APIs are mapped in a nearly 1:1 correspondence. For instance, a `CeedVector` object is exposed as `libceed.Vector` in Python, and may reference memory that is also accessed via Python arrays from the NumPy [vCV11] or Numba [LPS15] packages, for handling host or device memory (when interested in GPU computations with CUDA). Flexible pointer handling in libCEED makes it easy to provide zero-copy host and (GPU) device support for any desired Python array container.

libCEED's API

As illustrated in the Introduction, it is favorable to minimize memory motion, especially when computations are performed in parallel computing environments. In Finite Element codes that exploit data parallelism, the action of the operator can be described as *global*, when the operator is applied to data distributed across different nodes or compute devices, or *local*, when operating on a single portion of the data partition. libCEED's API provides the local action of an operator (linear or nonlinear) without assembling its sparse representation. The purely algebraic nature of libCEED allows efficient operator evaluations (selectable at runtime) and offers matrix-free preconditioning ingredients. While libCEED's focus is on high-order finite elements, the approach with which it is designed is algebraic and thus applicable to other discretizations in factored form. This algebraic decomposition also presents the benefit that it can equally represent linear or non-linear finite element operators.

Let us define the global operator as

$$A = P^T \underbrace{G^T B^T D B G}_{\text{libCEED's scope}} P, \tag{1}$$

where P is the parallel process decomposition operator (external to libCEED, which needs to be managed by the user via external packages, such as `petsc4py` [BAA+20], [DPKC11]) in which the degrees of freedom (DOFs) are scattered to and gathered from the different compute devices. The operator denoted by $A_L = G^T B^T D B G$ gives the local action on a compute node or process, where G is a local element restriction operation that localizes DOFs based on the elements, B defines the action of the basis functions (or their gradients) on the nodes, and D is the user-defined pointwise function describing the physics of the problem at the quadrature points, also called the QFunction (see Fig. 3). Instead of forming a single operator using a sparse matrix representation, libCEED composes the different parts of the operator described in equation (1) to apply the action of the operator $A_L = G^T B^T D B G$ in a fashion that is tuned for the different compute devices, according to the backend selected at run time.

In libCEED's terminology, the global or total vector is called a T-vector (cf. Fig. 3). This stores the true degrees of freedom of the problem. In a T-vector, each unknown has exactly one copy, on exactly one processor, or rank. The process decomposition, denoted by P in equation (1), is a non-overlapping partitioning. The application of the operator P to a T-vector results in an L-vector, or local vector. This stores the data owned by each rank. In an L-vector, each unknown has exactly one copy on each processor that owns an element containing it. This is an overlapping vector decomposition with overlaps only across different processors—there is no duplication of unknowns on a single processor. The nodes adjacent to different elements (at element corners or edges) will be the one that have more than one copy, on different processors. Applying an element restriction operator, denoted by G in equation (1), to an L-vector creates an E-vector. This stores the nodes grouped by the elements they belong to. In fact, in an E-vector each unknown has as many copies as the number of elements that contain it. The application of a basis operator B to an E-vector returns a Q-vector. This has the same layout of an E-vector, but instead of holding the different unknown values, a Q-vector stores the values at quadrature points, grouped by element.

The mathematical formulation of QFunctions, described in weak form, is fully separated from the parallelization and meshing concerns. In fact, QFunctions, which can either be defined by the user or selected from a gallery of available built-in functions in the library, are pointwise functions that do not depend on element resolution, topology, or basis degree (selectable at run time). This easily allows hp-refinement studies (where h commonly denotes the average element size and p the polynomial degree

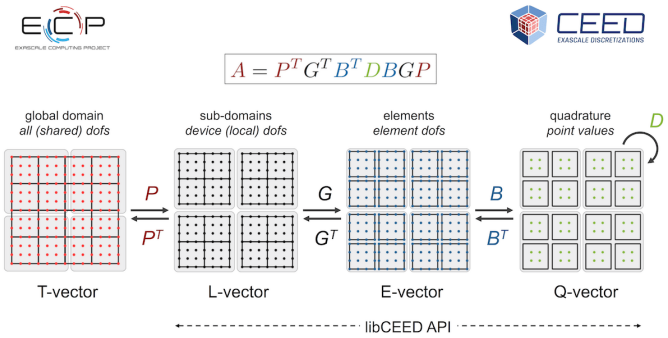


Fig. 3: *Operator decomposition.*

of the basis functions in 1D) and *p*-multigrid solvers. libCEED also supports composition of different operators for multiphysics problems and mixed-element meshes (see Fig. 4). Currently, user-defined QFunctions are written in C and must be precompiled as a foreign function library and loaded via `ctypes`. The single-source C QFunctions allow users to equally compute on CPU or GPU devices, all supported by libCEED. The ultimate goal is for users to write only Python code. This will be achieved in the near future by using the Numba high-performance Python compiler or Google's extensible system for composable function transformations, JAX [BFH+18], which can use just-in-time (JIT) compilation to compile for coprocessors and speed-up executions when sequences of operations are performed.

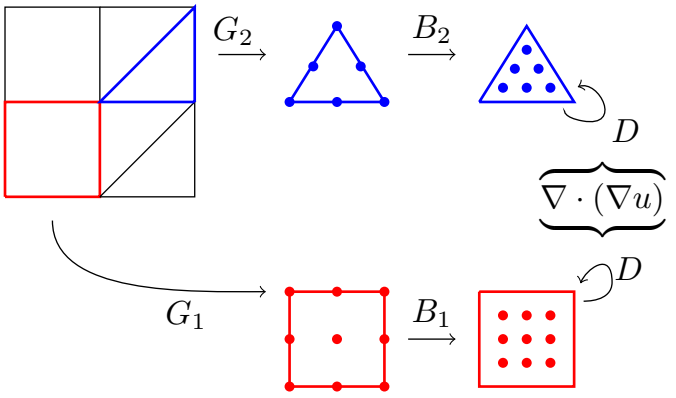


Fig. 4: *A schematic of element restriction and basis applicator operators for elements with different topology. This sketch shows the independence of QFunctions (in this case representing a Laplacian) element resolution, topology, or basis degree.*

Source Code Examples

LibCEED for Python is distributed through PyPI [PyP20] and can be easily installed via

```
$ pip install libceed
```

or

```
$ python -m pip install libceed
```

The package can then be simply imported via

```
>>> import libceed
```

The simple declaration of a `libceed.Ceed` instance, with default resource (`/cpu/self`) can be obtained as

```
>>> ceed = libceed.Ceed()
```

If libCEED is built with GPU support, the user can specify a GPU backend, e.g., `/gpu/occa` or `/gpu/cuda/gen`, with

```
>>> ceed = libceed.Ceed('/gpu/cuda/gen')
```

Next, we show the creation of a `libceed.Vector` of a specified size

```
>>> n = 10
>>> x = ceed.Vector(n)
```

Similarly, this could have been achieved by running

```
>>> x = ceed.Vector(size=10)
```

In the following example, we associate the data stored in a `libceed.Vector` with a `numpy.array` and use it to set and read the `libceed.Vector`'s data

```
>>> import numpy as np

>>> x = ceed.Vector(size=3)

>>> a = np.arange(1, 4, dtype="float64")
>>> x.set_array(a, cmode=libceed.USE_POINTER)

>>> with x.array_read() as b:
...     print(b)
...
[1. 2. 3.]
```

Similarly, we can set all entries of a `libceed.Vector` to the same value (e.g., 10) via

```
>>> x.set_value(10)
```

If the user has installed libCEED with CUDA support and Numba, they can use device memory for `libceed.Vectors`. In the following example, we create a `libceed.Vector` with a libCEED context that supports CUDA, associate the data stored in a `CeedVector` with a `numpy.array`, and get a Numba `DeviceNDArray` containing the data on the device.

```
>>> ceed_gpu = libceed.Ceed('/gpu/cuda')

>>> n = 4
>>> x = ceed_gpu.Vector(n)

>>> a = np.arange(1, n + 1, dtype="float64")
>>> x.set_array(a, cmode=libceed.USE_POINTER)

>>> with x.array_read(memtype=libceed.MEM_DEVICE) as
        device_array:
...     print(device_array)
...
[1. 2. 3. 4.]
```

Among the Finite Elements objects needed to compose an operator, in the following example we illustrate the creation and apply action of an element restriction, denoted by G in equation (1)

```
>>> ne = 3

>>> x = ceed.Vector(ne+1)
>>> a = np.arange(10, 10 + ne+1, dtype="float64")
>>> x.set_array(a, cmode=libceed.USE_POINTER)

>>> ind = np.zeros(2*ne, dtype="int32")
>>> for i in range(ne):
...     ind[2*i+0] = i
...     ind[2*i+1] = i+1
...
>>> r = ceed.ElemRestriction(ne, 2, 1, 1, ne+1, ind,
... cmode=libceed.USE_POINTER)

>>> y = ceed.Vector(2*ne)
>>> y.set_value(0)

>>> r.apply(x, y)
```

```
>>> with y.array_read() as y_array:
...     print('y =', y_array)
...
y = [10. 11. 11. 12. 12. 13.]
```

An H^1 Lagrange basis in d dimensions can be defined with the following code snippet

```
>>> d = 1
>>> b = ceed.BasisTensorH1Lagrange(
... dim=d,    # topological dimension
... ncomp=1,  # number of components
... P=2,      # number of basis functions (nodes)
...           # per dimension
... Q=2,      # number of quadrature points
...           # per dimension
... qmode=libceed.GAUSS_LOBATTO)
>>> print(b)
CeedBasis: dim=1 P=2 Q=2
      qref1d:          -1.00000000      1.00000000
   qweight1d:           1.00000000      1.00000000
    interp1d[0]:        1.00000000      0.00000000
    interp1d[1]:        0.00000000      1.00000000
      grad1d[0]:       -0.50000000      0.50000000
      grad1d[1]:       -0.50000000      0.50000000
```

In the following example, we show how to apply a 1D basis operator, denoted by B in equation (1), from an E-vector named `Ev`, to a Q-vector named `Qv`, and vice-versa, its transpose operator B^T

```
>>> Q = 4
>>> dim = 1
>>> Xdim = 2**dim
>>> Qdim = Q**dim
>>> x = np.empty(Xdim*dim, dtype="float64")
>>> for d in range(dim):
...     for i in range(Xdim):
...         x[d*Xdim + i] = 1 if (i % (2**(dim-d)))
... // (2**(dim-d-1)) else -1
...
>>> Ev = ceed.Vector(Xdim*dim)
>>> Ev.set_array(x, cmode=libceed.USE_POINTER)
>>> Qv = ceed.Vector(Qdim*dim)
>>> Qv.set_value(0)
>>> bx = ceed.BasisTensorH1Lagrange(dim, dim, 2, Q,
... libceed.GAUSS_LOBATTO)
>>> bx.apply(1, libceed.EVAL_INTERP, Ev, Qv)
>>> print(Qv)
CeedVector length 4
  -1.000000
  -0.447214
  0.447214
  1.000000

>>> bx.T.apply(1, libceed.EVAL_INTERP, Qv, Ev)
>>> print(Ev)
CeedVector length 2
-1.200000
1.200000
```

In the following example, we create two QFunctions (for the setup and apply, respectively, of the mass operator in 1D) from the gallery of available built-in QFunctions in libCEED

```
>>> qf_setup = ceed.QFunctionByName("Mass1DBuild")
>>> print(qf_setup)
Gallery CeedQFunction Mass1DBuild
  2 Input Fields:
    Input Field [0]:
      Name: "dx"
      Size: 1
      EvalMode: "gradient"
    Input Field [1]:
      Name: "weights"
      Size: 1
      EvalMode: "quadrature weights"
  1 Output Field:
    Output Field [0]:
      Name: "qdata"
      Size: 1
      EvalMode: "none"

>>> qf_mass = ceed.QFunctionByName("MassApply")
>>> print(qf_mass)
Gallery CeedQFunction MassApply
  2 Input Fields:
    Input Field [0]:
      Name: "u"
      Size: 1
      EvalMode: "interpolation"
    Input Field [1]:
      Name: "qdata"
      Size: 1
      EvalMode: "none"
  1 Output Field:
    Output Field [0]:
      Name: "v"
      Size: 1
      EvalMode: "interpolation"
```

The setup QFunction, named `qf_setup` in the previous example, is the one that defines the formulation of the geometric factors given by the correspondence between deformed finite element coordinates and reference ones. The apply QFunction, named `qf_mass` in the previous example, is the one that defines the action of the physics (in terms of the spatial discretization of the weak form of the PDE) the user wants to solve for. In this simple example, this represented the action of the mass matrix.

Finally, once all ingredients for a `libceed.Operator` are defined (i.e., element restriction, basis, and QFunction), one can create and apply a local operator as

```
>>> nelem = 15
>>> P = 5
>>> Q = 8
>>> nx = nelem + 1
>>> nu = nelem*(P-1) + 1

>>> # Vectors
>>> x = ceed.Vector(nx)
>>> x_array = np.zeros(nx)
>>> for i in range(nx):
...     x_array[i] = i / (nx - 1.0)
...
>>> x.set_array(x_array, cmode=libceed.USE_POINTER)
>>> qdata = ceed.Vector(nelem*Q)
>>> u = ceed.Vector(nu)
>>> v = ceed.Vector(nu)

>>> # Restrictions
>>> indx = np.zeros(nx*2, dtype="int32")
>>> for i in range(nx):
...     indx[2*i+0] = i
...     indx[2*i+1] = i+1
...
>>> rx = ceed.ElemRestriction(nelem, 2, 1, 1, nx, indx,
... cmode=libceed.USE_POINTER)
>>> indu = np.zeros(nelem*P, dtype="int32")
>>> for i in range(nelem):
...     for j in range(P):
...         indu[P*i+j] = i*(P-1) + j
...
>>> ru = ceed.ElemRestriction(nelem, P, 1, 1, nu, indu,
... cmode=libceed.USE_POINTER)
>>> strides = np.array([1, Q, Q], dtype="int32")
>>> rui = ceed.StridedElemRestriction(nelem, Q, 1,
... Q*nelem, strides)

>>> # Bases
>>> bx = ceed.BasisTensorH1Lagrange(1, 1, 2, Q,
... libceed.GAUSS)
>>> bu = ceed.BasisTensorH1Lagrange(1, 1, P, Q,
... libceed.GAUSS)
```

```
>>> # QFunctions
>>> qf_setup = ceed.QFunctionByName("Mass1DBuild")
>>> qf_mass = ceed.QFunctionByName("MassApply")

>>> # Setup operator
>>> op_setup = ceed.Operator(qf_setup)
>>> op_setup.set_field("dx", rx, bx,
... libceed.VECTOR_ACTIVE)
>>> op_setup.set_field("weights",
... libceed.ELEMRESTRICTION_NONE, bx,
... libceed.VECTOR_NONE)
>>> op_setup.set_field("qdata", rui,
... libceed.BASIS_COLLOCATED,
... libceed.VECTOR_ACTIVE)
>>> print('Setup operator: ', op_setup)
Setup operator:  CeedOperator
  3 Fields
  2 Input Fields:
    Input Field [0]:
      Name: "dx"
      Active vector
    Input Field [1]:
      Name: "weights"
      No vector
  1 Output Field:
    Output Field [0]:
      Name: "dx"
      Collocated basis
      Active vector

>>> # Apply Setup operator
>>> op_setup.apply(x, qdata)
```

For all of the illustrated classes of objects, `libceed.Ceed`, `libceed.Vector`, `libceed.ElemRestriction`, `libceed.Basis`, `libceed.QFunction`, and `libceed.Operator`, libCEED's Python interface provides a representation method so that they can be viewed/printed by simply typing

```
>>> print(x)
```

These and other examples can be found in the suite of Project Jupyter [KRKP+16] tutorials provided with libCEED in a Binder [lib20a] interactive environment, accessible on libCEED's development site [lib20b]. Finally, examples of integration of libCEED with other packages in the co-design Center for Efficient Exascale Discretizations (CEED), such as PETSc, MFEM, and Nek5000, can be found in the CEED distribution, which provides the full CEED software ecosystem [BAB+19], [KFA+20].

Conclusions

We have presented libCEED, a new lightweight, open-source, matrix-free Finite Element library, its conceptual framework, and new Python interface. libCEED's purely algebraic framework can unobtrusively be integrated in new and legacy software to provide performance portable applications. In this work, we have demonstrated the usage of libCEED's Python interface by providing examples of the creation and application of the main classes in libCEED's API: `libceed.Ceed`, `libceed.Vector`, `libceed.ElemRestriction`, `libceed.Basis`, `libceed.QFunction`, and `libceed.Operator`. We have showed how libCEED's simple interface allows for easy and composable library reuse and can open up new strategies for parallelism and scaling in high-performance Python.

Acknowledgments

The libCEED library is distributed under a BSD 2-Clause License with Copyright (c) 2017 of the Lawrence Livermore National Security, LLC. The work presented in this paper is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U. S. Department of Energy Organizations (the Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

References

[BAA+20] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.13, Argonne National Laboratory, 2020.

[BAB+19] Jed Brown, Ahmad Abdelfattah, Valeria Barra, Veselin Dobrev, Yohann Dudouit, Paul Fischer, Tzanio Kolev, David Medina, Misun Min, Thilina Ratnayaka, Cameron Smith, Jeremy Thompson, Stanimire Tomov, Vladimir Tomov, and Tim Warburton. CEED ECP Milestone Report: Public release of CEED 2.0, 2019. doi:10.5281/zenodo.2641316.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

[Bro10] Jed Brown. Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D. *Journal of Scientific Computing*, 45, October 2010. doi:10.1007/s10915-010-9396-8.

[CUD20] https://developer.nvidia.com/about-cuda, 2020. URL: https://developer.nvidia.com/about-cuda.

[DPKC11] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34:1124 – 1139, 2011. New Computational Methods and Software Tools. doi:https://doi.org/10.1016/j.advwatres.2011.04.013.

[KFA+20] Tzanio Kolev, Paul Fischer, Ahmad Abdelfattah, Shreyas Ananthan, Valeria Barra, Natalie Beams, Ryan Bleile, Jed Brown, Robert Carson, Jean-Sylvain Camier, Matthew Churchfield, Veselin Dobrev, Jack Dongarra, Yohann Dudouit, Ali Karakus, Stefan Kerkemeier, YuHsiang Lan, David Medina, Elia Merzari, Misun Min, Scott Parker, Thilina Ratnayaka, Cameron Smith, Michael Sprague, Thomas Stitt, Jeremy Thompson, Ananias Tomboulides, Stanimire Tomov, Vladimir Tomov, Arturo Vargas, Tim Warburton, and Kenneth Weiss. CEED ECP Milestone Report: Improve performance and capabilities of CEED-enabled ECP applications on Summit/Sierra, 2020. doi:10.5281/zenodo.3860804.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter Notebooks – a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing*, pages 87 – 90, 2016. doi:10.3233/978-1-61499-649-1-87.

[LF05] James W. Lottes and Paul F. Fischer. Hybrid multigrid/schwarz algorithms for the spectral element method. *Journal of Scientific Computing*, 24, 2005. doi:10.1007/s10915-004-4787-3.

[lib20a] libCEED Binder, 2020. URL: https://mybinder.org/v2/gh/CEED/libCEED/master?urlpath=lab/tree/examples/tutorials/tutorial-0-ceed.ipynb.

[lib20b] libCEED development site, 2020. URL: https://github.com/ceed/libceed.

[LIB20c] LIBXSMM development site, 2020. URL: http://github.com/hfp/libxsmm.

[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*, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2833157.2833162.

[MAG20] MAGMA development site, 2020. URL: https://bitbucket.org/icl/magma.

[MFE20] MFEM: Modular Finite Element Methods Library, 2020. URL: https://mfem.org/, doi:10.11578/dc.20171025.1248.

[Nek20] Nek5000, 2020. URL: https://nek5000.mcs.anl.gov/.

[OCC20] OCCA development site, 2020. URL: http://github.com/libocca/occa.

[Ors80] Steven A Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37:70 – 92, 1980. doi:https://doi.org/10.1016/0021-9991(80)90005-4.

[PyP20] The Python Package Index (PyPI), 2020. URL: https://pypi.org/.

[vCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13:22 – 30, 2011. doi:10.1109/MCSE.2011.37.

Spectral Analysis of Mitochondrial Dynamics: A Graph-Theoretic Approach to Understanding Subcellular Pathology

Marcus Hill[‡], Mojtaba Fazli[‡], Rachel Mattson[‡‡], Meekail Zain[‡‖], Andrew Durden[‡], Allyson T Loy[¶], Barbara Reaves[**], Abigail Courtney[¶], Frederick D Quinn[**], S Chakra Chennubhotla[††], Shannon P Quinn[‡§*]

Abstract—Perturbations of organellar structures within a cell are useful indicators of the cell's response to viral or bacterial invaders. Of the various organelles, mitochondria are meaningful to model because they show distinct migration patterns in the presence of potentially fatal infections, such as tuberculosis. Properly modeling and assessing mitochondria could provide new information about infections that can be leveraged to develop tests, treatments, and vaccines. Traditionally, mitochondrial structures have been assessed via manual inspection of fluorescent microscopy imagery. However, manual microscopy studies are labor-intensive and fail to provide a high-throughput for screenings. Thus, demonstrating the need for techniques that are more automated and utilize quantitative metrics for analysis. Yet, modeling mitochondria is no trivial task; mitochondria are amorphous, spatially diffuse structures that render traditional shape-based, parametric modeling techniques ineffective. We address the modeling task by using OrNet (Organellar Networks), a Python framework that utilizes probabilistic, graph-theoretic techniques to cast mitochondrial dynamics in the mold of dynamic social networks. We propose quantitative temporal and spatial anomaly detection techniques that leverage the graph connectivity information of the social networks to reveal time points of anomalous behavior and spatial regions where organellar structures undergo significant morphological changes related to a relevant change in environment or stimulus. We demonstrate the advantages of these techniques with the results of exhaustive graph-theoretic analyses over time in three different mitochondrial conditions. This methodology provides the quantification, visualization, and analysis techniques necessary for rigorous spatiotemporal modeling of diffuse organelles.

Introduction

Morphological perturbations of organellar structures inside cells are useful for characterizing infection patterns and, ultimately, developing therapies. In particular, tuberculosis, an infectious disease caused by *Mycobacterium tuberculosis* (Mtb), induces distinct structural changes of the mitochondria in invaded cells [FCGQR15]. This is significant because tuberculosis is responsible for approximately 1.5 million human fatalities annually, with growing resistance to current antibacterial treatment regimens [FCGQR15]. Studying the permanent changes in subcellular structures pre- versus post-infection will set the stage for genetic screens, whereby these changes can be studied under different mutations of the Mtb pathogen; with enough such studies and a subsequent understanding of how the Mtb pathogen affects its host, we can leverage that knowledge to develop tests, treatments, and vaccines.

Prior works have shown that the Mtb pathogen alters the shape of mitochondrial structures to disrupt vital functions provided by the organelle so that it can successfully invade a host [Dub16], [CAA18]. Modeling these perturbed subcellular structures for analysis is difficult because mitochondria are amorphous, spatially diffuse structures whose morphology exists within a dynamic continuum, ranging from fragmented individual mitochondrion to complex interconnected networks [FS12]. The morphology of mitochondria transitions between many states along its spectrum, as a result of fission and fusion events [SBM+08], [SNY08] and the observation of these morphological changes is referred to as mitochondrial dynamics [FS12]. Thus, understanding mitochondrial dynamics is useful for gaining insight regarding a host's response to infections and cellular invasions. Figure 1 depicts the morphological changes of mitochondria in two different cells that underwent either fission or fusion to illustrate mitochondrial dynamics.

Early approaches to assessing mitochondrial dynamics involved manually observing fission and fusion events in live microscopy imagery. A notable early study tagged two distinct groups of mitochondria with red and green fluorescent proteins before introducing the two groups together in the presence of polyethylene glycol (PEG) to induce fusion, then manually observed the resultant heterogeneous fluorescent structures to understand mitochondrial dynamics [LLFR02]. However, manual microscopy studies are labor-intensive and fail to provide a high-throughput for screenings [FS12]. These shortcomings have motivated many to research methodologies that are more automated by quantitatively modelling and assessing live microscopy imagery

‡ *Department of Computer Science, University of Georgia, Athens, GA 30602 USA*
‡‡ *Department of Cognitive Science, University of Georgia, Athens, GA 30602 USA*
‖ *Department of Mathematics, University of Georgia, Athens, GA 30602 USA*
¶ *Department of Microbiology, University of Georgia, Athens, GA 30602 USA*
** *Department of Infectious Diseases, University of Georgia, Athens, GA 30602 USA*
†† *Department of Computational and Systems Biology, University of Pittsburgh, Pittsburgh, PA 15232 USA*
* *Corresponding author: spq@uga.edu*
§ *Department of Cellular Biology, University of Georgia, Athens, GA 30602 USA*

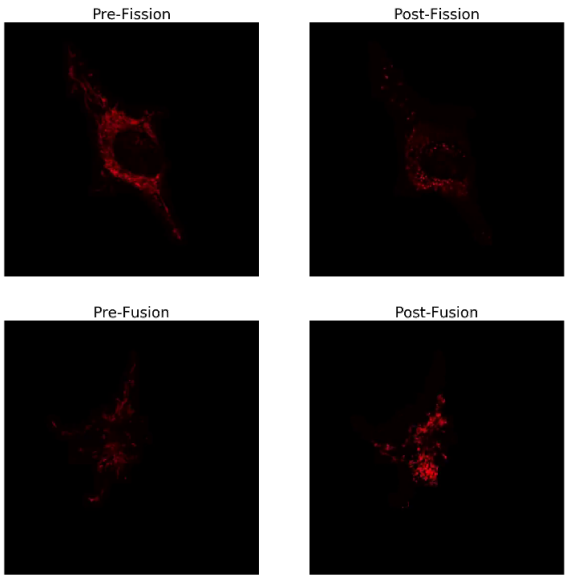


Fig. 1: *Frames from two different live microscopy videos depicting the mitochdonria before and after morphological events.*

of mitochondria [ADATLBR+18], [SBM+08], [MLS10].

Assessment of mitochondrial dynamics via live microscopy has been studied from various scientific perspectives. Most studies utilized confocal fluorescent microscopy to capture the morphological changes of the mitochondria [SCE+17]. One imagery-based approach proposed a quantitative methodology that measured the lengths of all mitochondria present in a cell, both prior to and post the occurrence of either a fission or fusion event, to determine whether the mitochondria fused or fragmented [SBM+08]. Limitations of that approach was that it required manually denoting regions of interest to assess only a subset of the mitochondria present, and it intentionally excluded any mitochondria located in dense clusters because of the difficulty in determining the precise shapes and dimensions of individual mitochondrion. Another quantitative approach also leveraged confocal microscopy imagery to utilize a technique known as FRAP, or fluorescence recovery after photobleaching, to assess mitochondrial dynamics [MLS10]. This specific FRAP approach involved bleaching the cell in a designated region and monitoring the recovery of fluorescence as fluorescently tagged mitochondria migrate from unbleached areas to the bleached zones [MLS10]. However, such an approach can be jeopardized by the motility of mitochondria and its environment; unexpected movement from mitochondria, or even the entire cell, can disrupt a FRAP analysis requiring the sample being assessed to be discarded. Both of these early approaches risk overlooking crucial morphological information because only a subset of the mitochondria present in the cell can be used for analysis.

In more recent literature, a novel methodology was proposed that modeled the morphology of mitochondria by casting all local diffuse clusters of mitochondria present in a cell as nodes within an evolving graph, known as a dynamic social network [ADATLBR+18]. Dynamic social networks are well-suited for modeling mitochondria because the granularity of the clusters being modeled can be adjusted by increasing or decreasing the number of nodes used. These networks overcome the limitations of prior approaches because they do not require any manual intervention nor are they negatively affected by organellar motility. Additionally, this approach does not seek to assess only specific well-behaved mitochondria, but any that are visible in clusters around the cell regardless of their morphological state (i.e. fragmented, fused, etc.). Our work seeks to elucidate mitochondrial dynamics by providing quantitative methodologies to measure spatial and temporal regions of anomalous morphological behavior via spectral analysis of dynamic social networks.

Data Acquisition

In our efforts to demonstrate the morphological spectrum that mitochondria undergo, we have amassed a collection of confocal imaging videos of live HeLa cells fluorescently tagged with the protein DsRed2-Mito-7. We maintained three distinct groups of cells: a group that was not exposed to any external stimulant, referred to as our control group; a group that was exposed to listeriolysin O (llo), a pore-forming toxin, to induce mitochondrial fragmentation; and a group that was exposed to mitochondrial-division inhibitor 1 (mdivi) to induce mitochondrial fusion. Live imaging videos of each cell was recorded with a Nikon A1R confocal microscope. The imaging occurred in an environment that maintained 37 degrees celsius and 5% CO2. Every imaging video consists of at least 20,000 frames, of dimensions 512x512, captured at 100 frames per second. In all of our imagery, each red "dot" depicts a single mitochondrion within a cell. For scale, the length of mitochondria is typically between 500 nm to 1 mm or greater, and the average diameter is approximately 500 nm [MLS10], [DC07].

Spectral Analysis of Social Networks

Mitochondrial structures can respond in drastic, unpredictable ways to an environmental change or an external stimulus, and our work seeks to characterize these responses from both the spatial and temporal contexts. We were able to explore those perspectives by analyzing microscopy imagery, primarily videos, of fluorescently tagged live HeLa cells post-exposure to drug treatments that induced either fusion or fragmentation of the mitochondria in the cells [ADATLBR+18]. We modeled and analyzed the mitochondria using OrNet (Organellar Networks), an open-source Python framework built on libraries within the scientific Python ecosystem that models subcellular organelles as dynamic social networks [FHD+20].

OrNet utilizes a probabilistic approach, involving Gaussian mixture models (GMMs), to construct mitochondrial cluster graphs [ADATLBR+18], [FHD+20]. GMMs were utilized to determine spatial regions of the microscopy imagery that corresponded to the mitochondrial clusters by iteratively updating the parameters of underlying mixture distributions until they converged. This approach assumes that the spatial locations of mitochondria are normally distributed with respect to their associated clusters [ADATLBR+18]. The post-convergence parameters of the mixture distributions, specifically the means and covariances, were then used for constructing the social network graph. The means corresponded to the center spatial coordinates of mitochondrial clusters, and for this reason they were selected to be the nodes in the graphs. The edges, which represent the relationships between clusters, were defined by the Hellinger distance between the respective mixture distributions. This modeling process occurred

for every frame in a microscopy video; therefore, each frame updates the state of the network's graph at a discrete point in time. Traced over time, the dynamics of the social networks, appearing as perturbations in connected nodes via changes in the edge weights, OrNet tracks the changes of the spatial relationships between mitochondrial clusters.

By modeling the spatiotemporal relationships of mitochondria as a dynamic social network, the graph states could be represented as Laplacian matrices. A Laplacian matrix is a useful representation of a graph that enables the analysis of its properties via spectral graph theory techniques. Eigendecomposition, or the factorization of a matrix into its eigenvalues and eigenvectors, is a graph theoretic technique that is the cornerstone of our proposed methodologies. Eigendecomposition of a graph Laplacian yields vital information about the connectedness of that graph [CGotMS97]. In the context of mitochondrial dynamics, the connectedness of a graph provides a quantitative description about the morphology at a given time. By leveraging such quantitative descriptions, our techniques are able to indicate spatial and temporal regions demonstrating anomalous behavior.

Temporal Anomaly Detection

Detecting when morphology-altering events occur is an important aspect to understanding mitochondrial dynamics. Temporal indicators of organellar activity improve qualitative assessments of microscopy imagery by eliminating the need to manually inspect every frame, only those that immediately precede or succeed an anomalous event. Additionally, the effects of local events on the global mitochondrial structure are more distinct. This process of indicating time points when distinct organellar activity is occurring is a temporal anomaly detection task. We addressed this task by utilizing the graph connectivity information provided by the eigenvalue vectors to detect anomalous behaviors.

Eigendecomposition of each mitochondrial cluster graph that comprises the dynamic social network results in a number of eigenvalue vectors and eigenvector matrices that correspond to the number of graph states in the network. Because these vectors and matrices have a natural ordering, the information is essentially a time series dataset. We highlight anomalous time points in the data by first computing the average of each eigenvalue vector, then indicating time points whose averages are statistical outliers. Outliers are determined by computing the z-score, or standard score, for every time point based on the distance between the average of its associated eigenvalue vector and the mean of a few preceding averages; if the distance exceeds some threshold value, typically two standard deviations, then it is considered an outlier. The number of preceding averages used is predetermined by a fixed window size. This sliding window approach enables adaptive thresholding values to be computed for declaring anomalous behavior that are derived from local morphological events, rather than a fixed global constant.

In essence, this approach utilizes the eigenvalues to characterize the magnitude of spatial transformations experienced by the morphology. Therefore, morphology-altering events, like fission and fusion, are likely to be discovered by highlighting time points where eigenvalue vectors are demonstrating anomalous behavior.

The Python code we utilized to perform temporal anomaly detection is below: this function computes anomalous time points and displays the subsequent eigenvalue time-series and outlier signal plots. The parameters to the function are the time-series of eigenvalue vectors that correspond to the dynamic social network, a window size, and a threshold value. An example of the plots generated by this code is shown in Figure 2.

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

def temporal_anomaly_detection(eigen_vals, window,
                               threshold):
 '''
 Generates a figure comprised of a time-series plot
 of the eigenvalue vectors, and an outlier detection
 signals plot.

 Parameters
 ----------
 eigen_vals: NumPy array (NXM)
     Matrix comprised of eigenvalue vectors.
     N represents the number of frames in the
     corresponding video, and M is the number of
     mixture components.
 window: int
     The size of the window to be used for anomaly
     detection.
 threshold: float
     Value used to determine whether a signal value
     is anomalous.

 Returns
 -------
 NoneType object
 '''
 eigen_vals_avgs = [np.mean(x) for x in eigen_vals]
 moving_avgs = np.empty(shape=(eigen_vals.shape[0],),
                        dtype=np.float)
 moving_stds = np.empty(shape=(eigen_vals.shape[0],),
                        dtype=np.float)
 z_scores = np.empty(shape=(eigen_vals.shape[0],),
                     dtype=np.float)
 signals = np.empty(shape=(eigen_vals.shape[0],),
                    dtype=np.float)

 moving_avgs[:window] = 0
 moving_stds[:window] = 0
 z_scores[:window] = 0
 for i in range(window, moving_avgs.shape[0]):
     moving_avgs[i] = \
         np.mean(eigen_vals_avgs[i - window:i])
     moving_stds[i] = \
         np.std(eigen_vals_avgs[i - window:i])
     z_scores[i] = \
         eigen_vals_avgs[i] - moving_avgs[i]

     z_scores[i] /= moving_stds[i]

 for i, score in enumerate(z_scores):
     if score > threshold:
         signals[i] = 1
     elif score < threshold * -1:
         signals[i] = -1
     else:
         signals[i] = 0

 sns.set()
 fig = plt.figure()
 ax = fig.add_subplot(211)
 ax.plot(eigen_vals)
 ax.set_ylabel('Magnitude')
 ax = fig.add_subplot(212)
 ax.plot(z_scores)
 ax.set_xlabel('Frame')
 ax.set_ylabel('Signal')
 plt.show()
 plt.close()
```

Fig. 2: *The top plot illustrates the eigenvalue time-series data of an llo cell that experienced a mitochondrial fission event, and the bottom plot shows the corresponding outlier signal plot. Peaks in the signal plot represent time points declared anomalous by the temporal anomaly detection technique.*

Spatial Anomaly Detection

After indicating discrete times points where the morphology experienced significant perturbations, quantitatively determining the spatial locations of significant structural changes is crucial for assessing mitochondrial dynamics. Mitochondria are spatially diffuse structures that occupy a vast amount of the cell and, as a result, many areas of the cell require detailed inspection to identify all significant spatial changes. However, many structural perturbations go unnoticed when evaluated with purely qualitative metrics because of the large search space and the inherent difficulty in tracking microscopic objects. Thus, we sought to provide a quantitative technique to indicate spatial regions demonstrating anomalous morphological behavior.

Anomalous morphological behavior can be defined as spatial regions shifting suddenly, or major structural changes taking place in the underlying social network: edges being dropped or formed, nodes appearing or disappearing. The process of tracking such regions is, in essence, an object detection task because specific mitochondrial clusters are being monitored as the global structure evolves over time. By treating this task as such, we utilized bounding boxes to highlight the regions of significance. The coordinates of the bounding boxes were computed based on the pixel coordinates denoted by the GMMs that corresponded to the spatial locations of the mitochondrial clusters. Therefore, a bounding box can be displayed for each mitochondrial cluster determined by the GMM. However, rendering every bounding box can obfuscate the regions demonstrating anomalous behavior because some of the mitochondrial cluster boundaries may overlap. As a result, we utilized only the most significant, non-overlapping regions for analysis.

Regions demonstrating the most significant amount of structural variance are determined via analysis of the eigenvector matrices. The number of eigenvector matrices corresponds with the number of graph states recorded in the social network. Each row in an eigenvalue matrix is related to a mixture distribution, and by extension a spatial region of the imagery. To determine the regions demonstrating the most amount of variance, the total euclidean distance of each row vector between graph states is computed. Ultimately, the spatial regions that corresponded to the eigenvector rows demonstrating the highest amounts of variance were selected as regions of interest to be highlighted by the bounding boxes.

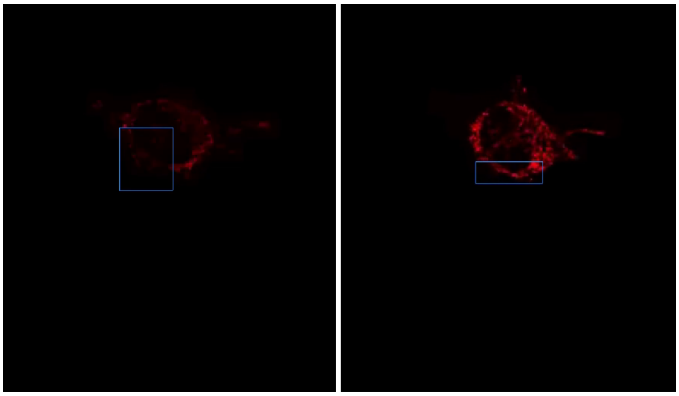


Fig. 3: *Image on the left shows the initial spatial location and size of a bounding box for a mitochondrial cluster from the first frame of a mdivi cell's microscopy video, which depicts a mitochondrial fusion event, and the image on the right shows the spatial location and size of a bounding box corresponding to the same cluster on the final frame of the video. This figure highlights the ability of our spatial anomaly detection technique to accurately track the mitochondria as it undergoes morphological transformations.*

Below is the code utilized to perform spatial anomaly detection: this function draws bounding boxes for the mitochondrial cluster regions in a microscopy video. The parameters for the function are the file path to the input video; means and covariance matrices from the GMM; the eigenvector matrices; an integer that indicates the maximum number of boxes to display; the path to the directory where the output video will be saved; and the number of standard deviations away from center spatial coordinates, in both dimensions, to construct the box boundaries. Figure 3 displays frames of a video generated by this code.

```
import imageio
import numpy as np

def spatial_anomaly_detection(vid_path, means,
                              covars, eigen_vecs, k,
                              outdir_path,
                              std_threshold=3):
    '''
    Draws bounding boxes around the mixture component
    regions demonstrating the most variance.

    Parameters
    ----------
    vid_path: string
        Path to the input video.
    means: NumPy array (NxMx2)
        Pixel coordinates corresponding to the mixture
        component means. N is the number of video
        frames, M the number of mixture components,
        and 2 denotes the 2D pixel coordinate.
    covars: NumPy array (NxMx2x2)
        Covariance matrices of the guassian mixture
        components. N is the number of video frames,
        M is the number of mixture components, and 2x2
        denotes the covariance matrix.
    eigen_vecs: NumPy array (NxMxM)
        Eigenvector matrix. N represents the number of
        frames in the corresponding video, M is the
        number of mixture components.
    k: int
        Number of the most significant non-overlapping
        regions to display bounding boxes for. The
        actual number may be less than k, if the video
        does not contain that many non-overlapping
        regions.
    outdir_path: string
        Directory path to save the bounding box video.
    std_threshold: float
        The number of standard deviations to use
        to compute the spatial region of the bounding
        box. Default is three.
    '''

    input_vid_title = os.path.split(vid_path)[1]
    out_vid_title = \
        input_vid_title.split('.')[0] + '.mp4'
    out_vid_path = os.path.join(outdir_path,
                                out_vid_title)
    with imageio.get_reader(vid_path) as reader, \
         imageio.get_writer(
            out_vid_path, mode='I', fps=1) as writer
        fps = reader.get_meta_data()['fps']
        size = reader.get_meta_data()['size']
        distances = \
            absolute_distance_traveled(eigen_vecs)
        descending_distances_indices = \
           np.flip(np.argsort(distances))
        region_indices = find_initial_boxes(
            means,
            covars,
            size,
            descending_distances_indices,
            k
        )
        num_of_boxes = len(region_indices)
        box_colors = \
            np.random.randint(
                256,
                size=(num_of_boxes),
                3,
            )

        for i, frame in enumerate(reader):
            current_frame = frame
            for index, j in enumerate(region_indices):

                row_bounds, col_bounds = \
                    compute_region_boundaries(
                        means, covars, size, i, j
                    )
                row_diff = row_bounds[1] - row_bounds[0]
                col_diff = col_bounds[1] - col_bounds[0]

                color = box_colors[index]
                current_frame[
                    row_bounds[0]:row_bounds[1],
                    col_bounds[0],
                    :,
                ] = color
                current_frame[
                    row_bounds[0]:row_bounds[1],
                    col_bounds[1],
                    :,
                ] = color
                current_frame[
                    row_bounds[0],
                    col_bounds[0]:col_bounds[1],
                    :,
                ] = color
                current_frame[
                    row_bounds[1],
                    col_bounds[0]:col_bounds[1],
                    :,
                ] = color

            writer.append_data(current_frame)
```

Experiments

We first evaluated the temporal anomaly detection methodology by plotting the eigenvalue time-series and outlier signal for each cell. For our experiments, we utilized a window size of 20 and a threshold value of 2. An example of the plots generated is shown in Figure 2. Next, we evaluated the video frames that corresponded with each anomalous time point in every video. In each frame, significant changes in the morphology are visible, especially in the llo and mdivi videos. This is meaningful because the morphology of mitochondria changes subtly between frames, making it a tedious task to manually determine when any important event occurred. However, the anomalous time points indicate specific video frames where morphological changes are visible: the anomalous llo video frames illustrate the fragmentation process by depicting the clusters at distinct times where they are visibly smaller, and conversely, the anomalous mdivi frames highlight times where the clusters are noticeably larger. To illustrate the process of temporally tracking morphological activity, Figure 4 displays all of the frames in a llo cell's microscopy video that correspond to time points declared anomalous by our temporal anomaly detection technique.

Unexpectedly, we noticed anomalous behavior was indicated in a subset of our control videos. This was not anticipated because the control cells were not exposed to any stimuli, and their mitochondrial structures did not display any significant changes during the duration of the videos. This phenomenon highlighted the sensitivity of our approach; any significant movement of the mitochondria, such as a sudden migration, is likely to be detected as an anomalous event. Therefore, the temporal indicators will denote frames where morphological events are occurring, but they should not be relied on solely for any behavioral inference regarding the mitochondria's morphology.

Our spatial anomaly detection methodology was evaluated by inspecting the regions highlighted by the bounding boxes in each cell type. The effectiveness of this approach was demonstrated through assessment of the llo and mdivi videos because mito-

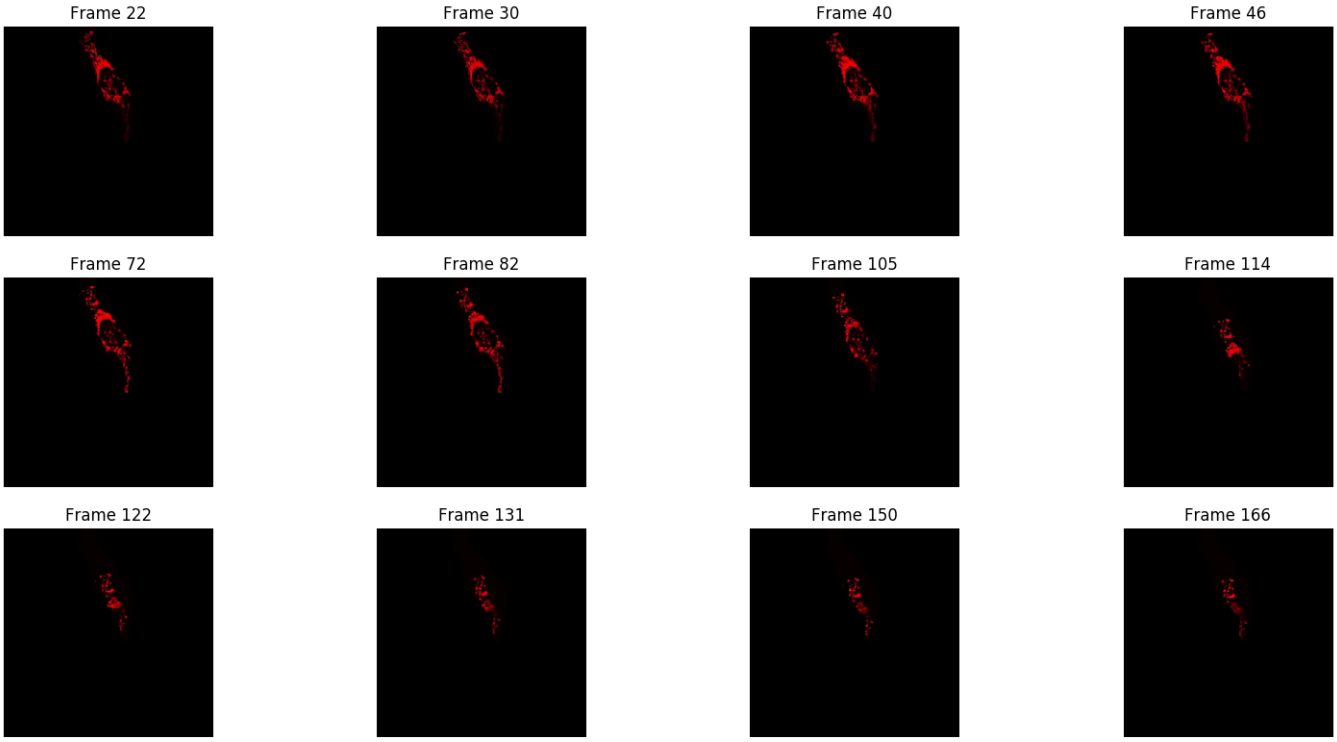


Fig. 4: *Frames from an llo video that were declared anomalous by the temporal anomaly detecton technique. These frames correspond to the same cell whose eigenvalue time-series and outlier signal plots were displayed in Figure 2.*

chondrial clusters of both types were displaced as their videos progressed. Mitochondria in the llo videos fragment and become much smaller, and in some instances this occurs until the clusters are no longer visible; in mdivi videos many of the smaller clusters merge with larger ones, effectively, making some regions of the cell no longer occupied by any mitochondrial structures. Yet, the bounding boxes were able to adapt accordingly to these spatial changes because the spatiotemporal relationships of clusters were captured within the dynamic social networks. The coordinates of the bounding boxes were computed using the parameters, specifically the mean and covariance, of the corresponding mixture distributions. As a result, the boxes were able to track the mitochondrial clusters as they moved around the cell or shrunk in size. In many cases, the clusters moved completely outside the area highlighted by initial bounding boxes, so the ability to adjust the shape and spatial locations of the boxes allows for the regions demonstrating anomalous behavior to always remain the areas being highlighted. Figure 3 depicts the spatial location and size of a bounding box corresponding to a mitochondrial region within a mdivi cell both before and after a fusion event occurs.

Discussion

Both the temporal and spatial anomaly detection methodologies have proven effective in quantitatively characterizing mitochondrial dynamics, however, the extent of their effectiveness is largely dependent on the selection of adequate parameters. For the temporal methodology the free parameters are the threshold value, window size. A threshold value too high will result in none of the time points being declared anomalous, while too low will result in a high number of frames being considered anomalous, even though the morphology may have only changed slightly between the time points. The window size is important for determining how distinct the current time point's eigenvalues are compared to those of the previous frames, and it behaves similarly to the threshold parameters: if the value is too high or low, the number of time points declared anomalous can change drastically. The spatial anomaly detection methodology has only one free parameter, the threshold value used to determine size of the bounding boxes. Due to the assumption that the spatial locations of mitochondria within each cluster are normally distributed [ADATLBR+18], we found that three standard deviations away from the mean, with respect to each dimension, is sufficient for a bounding box to encompass all the mitochondria that are members of the cluster being highlighted. Ultimately, these approaches are sensitive to the parameters selected, and the usage of adequate values can enhance the anomaly detection process.

Conclusion

The morphology of mitochondria is perturbed in distinct ways by the presence of bacterial or viral infections in the cell, and modeling these structural changes can aid in understanding both the infection strategies of the pathogen, and cellular re-

sponse. Modeling mitochondria poses many challenges because it is an amorphous, diffuse subcellular structure. Yet, dynamic social networks are well-suited for the task because they are capable of representing the global structure of mitochondria by flexibly modeling the many local clusters present in the cell. This extensible modeling approach enables the spatiotemporal relationships of the mitochondrial clusters to be explored using theoretic graph techniques. We proposed quantitative spatial and temporal anomaly detection methodologies that could be utilized in conjunction with traditional qualitative metrics to elucidate mitochondrial dynamics. We ultimately hope to use these spectral analytics and the OrNet software package to conduct large-scale genomic screens of *Mycobacterium tuberculosis* mutants, in an effort to build a deeper understanding of how the pathogen invades cells and induces cell death at the genetic level. This work is one of the first steps toward that ultimate goal.

REFERENCES

[ADATLBR+18] Andrew Durden, Allyson T Loy, Barbara Reaves, Mojtaba Fazli, Abigail Courtney, Frederick D Quinn, S Chakra Chennubhotla, and Shannon P Quinn. Dynamic Social Network Modeling of Diffuse Subcellular Morphologies. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 1 – 7, 2018. doi:10.25080/Majora-4af1f417-000.

[CAA18] Bridgette M Cumming, Kelvin W Addicott, and John H Adamson. Mycobacterium tuberculosis induces decelerated bioenergetic metabolism in human macrophages. *Biochemistry and Chemical Biology, Microbiology and Infectious Disease*, 2018. doi:10.7554/eLife.39169.

[CGotMS97] Fan R. K. Chung, Fan Chung Graham, and Conference Board of the Mathematical Sciences. *Spectral Graph Theory*. 1997.

[DC07] Scott A. Detmer and David C. Chan. Functions and dysfunctions of mitochondrial dynamics. *Nature Reviews Molecular Cell Biology*, 2007. doi:10.1038/nrm2275.

[Dub16] Rikesh K. Dubey. Assuming the role of mitochondria in mycobacterial infection. *International Journal of Mycobacteriology*, 2016. doi:10.1016/j.ijmyco.2016.06.001.

[FCGQR15] Kari Fine-Coulson, Steeve Giguere, Frederick D. Quinn, and Barbara J. Reaves. Infection of a549 human type ii epithelial cells with mycobacterium tuberculosis induces changes in mitochondrial morphology, distribution and mass that are dependent on the early secreted antigen, esat-6. *Microbes Infect.*, 2015. doi:10.1016/j.micinf.2015.06.003.

[FHD+20] Mojtaba Fazli, Marcus Hill, Andrew Durden, Rachel Mattson, Allyson T Loy, Barbara Reaves, Abigail Courtney, Frederick D Quinn, Chakra Chennubhotla, and Shannon Quinn. Ornet - a python toolkit to model the diffuse structure of organelles as social networks. *Journal of Open Source Software*, 2020. doi:10.21105/joss.01983.

[FS12] Andrew Ferree and Orian Shirihai. Mitochondrial dynamics: The intersection of form and function. *Advances in Experimental Biology and Medicine*, 2012. doi:10.1007/978-1-4614-3573-0_2.

[LLFR02] Frederic Legros, Anne Lombes, Paule Frachon, and Manuel Rojo. Mitochondrial fusion in human cells is efficient, requires the inner membrane potential, and is mediated by mitofusins. *Molecular Biology of the Cell*, 2002. doi:10.1091/mbc.E02-06-0330.

[MLS10] Kasturi Mitra and Jennifer Lippincott-Schwartz. Analysis of mitochondrial dynamics and functions using imaging approaches. *Curr Protoc Cell Biol*, 2010. doi:10.1002/0471143030.cb0425s46.

[SBM+08] Wenjun Song, Blaise Bossy, Ola J Martin, Andrew Hicks, Sarah Lubitz, Andrew B Knott, and Ella Bossy-Wetzel. Assessing mitochondrial morphology and dynamics using fluorescence wide-field microscopy and 3d image processing. *Methods*, 2008. doi:10.1016/j.ymeth.2008.10.003.

[SCE+17] Miguel Sison, Sabyasachi Chakrabortty, Jerome Extermann, Amir Nahas, Paul James Marchand, Antonio Lopez, Tanja Weil, and Teho Lasser. 3d time-lapse imaging and quantification of mitochondrial dynamics. *Scientific Reports*, 2017. doi:10.1038/srep43275.

[SNY08] Der-Fen Suen, Kristi L. Norris, and Richard J. Youle. Mitochondrial dynamics and apoptosis. *Genes and Development*, 2008. doi:10.1101/gad.1658508.

Matched Filter Mismatch Losses in MPSK and MQAM Using Semi-Analytic BEP Modeling

Mark Wickert‡*, David Peckham‡

Abstract—The focus of this paper is the bit error probability (BEP) performance degradation when the transmit and receive pulse shaping filters are mismatched. The modulation schemes considered are MPSK and MQAM. In the additive white Gaussian noise (AWGN) channel both spectral efficiency and noise mitigation is commonly achieved by using square-root raised cosine (SRC) pulse shaping at both the transmitter and receiver. The novelty of this paper primarily lies in the use semi-analytic BEP simulation for conditional error probability calculations, with transmit and receive filter mismatch, the optional inclusion of a small FIR equalizer. For lower order MPSK and MQAM, i.e., 8PSK and 16QAM E_b/N_0 power degradation at BEP $= 10^{-6}$ is 0.1 dB when the excess bandwidth mismatch tx/rx = 0.25/0.35 or 0.35/0.25, but quickly grows as the modulation order increases and/or the mismatch increases.

Index Terms—digital modulation, pulse shaping, phase-shift keying, quadrature amplitude modulation

Introduction

In the early days of satellite and space communications, digital modulation schemes focused on constant envelope waveforms, in particular phase-shift keying (PSK), with rectangular pulse shaping [Lindsey]. The need for spectral efficiency is ever present in modern communication systems [Ziemer], [Proakis], and [Goldsmith]. The use of pulse shaping makes spectral efficiency possible, at the expense of non-constant envelope waveforms [Ziemer]. Today m-ary PSK (MPSK) and high density m-ary quadrature amplitude modulation (MQAM), both with pulse shaping, are found in satellite communications as well as terrestrial communications, e.g., WiFi, cable modems, and 4th generation cellular via long term evolution (LTE). The term m-ary refers to the fact that bandwidth efficient signaling is accomplished using an M symbol alphabet of complex signal amplitudes, $c_k = a_k + jb_k$, to encode the transmission of a serial bit stream. In an m-ary digital modulation scheme we send $\log_2 M$ bits per symbol. The objective being to transmit more bits/s/Hz of occupied spectral bandwidth.

In certain applications the precise pulse shape used by the transmitter is not known by the receiver. The use of an equalizer is always an option in this case, but it adds complexity and for burst mode systems, e.g., TDMA, the convergence time of an adaptive equalizer is another issue to deal with.

The focus of this paper is the bit error probability (BEP) performance degradation when the transmit and receive pulse shaping filters are mismatched. The modulation schemes considered are MPSK and MQAM. In the additive white Gaussian noise (AWGN) channel both spectral efficiency and noise mitigation is commonly achieved by using square-root raised cosine (SRC) pulse shaping at both the transmitter and receiver. The system block diagram is shown in Figure 1. The parameters α_{tx} and α_{rx} control the excess bandwidth factors of the transmit and received filters respectively. Notice that this block diagram also shows a symbol-spaced equalizer to allow for the exploration of potential performance improvement, subject to how much complexity can be afforded, and the need for rapid adaptation in the case of burst mode transmission. We take advantage of semi-analytic simulation techniques described in [Tranter] to allow fast and efficient performance evaluation. All of the simulation software is written in open-source Python using classes to encapsulate the computations, waveform parameters, and calculation results. The mathematical foundation is statistical decision theory, which in the machine learning would be multiclass classification with apriori decision boundaries.

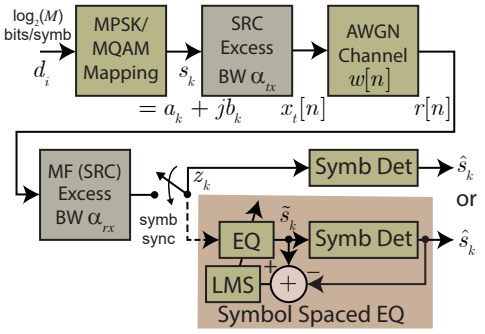


Fig. 1: *System top level block diagram showing transmit and receive pulse shaping filters with mismatch, i.e., $\alpha_{tx} \neq \alpha_{rx}$, and optionally the inclusion of an adaptive equalizer.*

Other authors such as, [Harris] and [Xing], have made mention of matched filter mismatch, but not in the same context as we consider in this paper. Harris is primarily driven by sidelobe reduction from the transmitter perspective, while Xing wishes to avoid the added complexity of an equalizer by using a specially designed receiving filter. Here we are concerned with the situation where the receiver does not always know the exact design of the transmit pulse shape filter, in particular the excess bandwidth factor.

The remainder of this paper is organized as follows. We

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

first provide a brief tutorial on digital communications at the waveform level. Next we consider residual errors at the matched filter output when using a simple truncated square-root raised cosine (SRC) finite impulse response (FIR). In particular we consider filter lengths of $\pm L$ symbols in duration. We then briefly explain how a symbol-spaced adaptive equalizer can be inserted at the output of the matched filter to compensate for pulse shape mismatch. We then move on to briefly review the concept of semi-analytic (SA)simulation and the develop conditional error probability expressions for MPSK and MQAM. Finally, we move into performance results.

Characterizing Digital Communications at the Waveform Level

To provide more context for the theoretical development of the semi-analytic simulation technique used in this paper and prepare for the system performance characterization of the results section, we now consider three common digital communication characterization techniques: *IQ Diagrams*, *eye diagrams*, and *bit error probability* (BEP) versus received signal energy-to-noise power spectral density (E_b/N_0) curves.

IQ Diagrams

An IQ diagram is a representation of a signal modulated by a digital modulation scheme such as MQAM or MPSK. It displays the signal as a two-dimensional *xy*-plane scatter diagram in the complex plane at symbol sampling instants. The angle of a point, measured counterclockwise from the horizontal axis, represents the phase shift of the carrier wave from a reference phase. The distance of a point from the origin represents a measure of the amplitude or power of the signal. The number of IQ points in a diagram gives the size of the *alphabet* of symbols that can be transmitted by each sample, and so determines the number of bits transmitted per sample. For the purposed of this paper it will be a power of 2. A diagram with four points, for example, represents a modulation scheme that can separately encode all 4 combinations of two bits: 00, 01, 10, and 11 and so can transmit two bits per sample. Figure 2 shows an 8-PSK IQ Diagram.

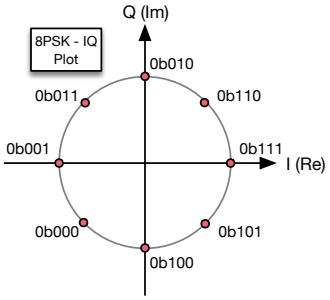


Fig. 2: *8-PSK IQ Diagram shows information is transmitted as one of 8 symbols, each representing 3 bits of data.*

Eye Diagrams

An eye diagram is a tool for the evaluation of the combined effects of channel noise and inter-symbol interference (ISI) on the performance of a channel. Several system performance measures can be derived by analyzing this display. If the signals are poorly synchronized with the system clock, or filtered improperly, or too noisy, this can be observed from the eye diagram. An open eye pattern corresponds to minimal signal distortion (clear diamond shape in the left plot). Distortion of the signal waveform due to ISI and noise appears as closure of the eye pattern (note partial closure on the right plot). The tight waveform bundles at the maximum opening correspond to tight scatter points of the IQ diagram, i.e., similar to the opposing pair of points on the real axis of Figure 2. Since the waveform is complex there is an eye diagram for both the real part and the imaginary part of the signal. For the purposes of this paper we will be looking at the closure of the eye pattern as the mismatch of the filters increases, similar to moving from the left to right side of Figure 3.

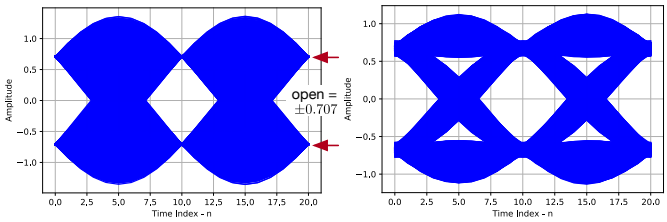


Fig. 3: *4PSK eye diagram: perfect channel (left), channel distortions present (right), both assuming 10 samples per symbol.*

Bit Error Probability (BEP) Curves

In digital transmission, the number of bit errors is the number of received bits over a communication channel that have been altered due to noise, interference, distortion (improper filtering), carrier phase tracking errors, and bit synchronization errors. The bit error probability (BEP), P_e, in a practical setting is the number of bit errors divided by the total number of transferred bits during a studied time interval. The BEP curves are plotted as $\log_{10} P_e$ versus the received energy per bit-to-noise power spectral density ratio in dB, i.e., $10\log_{10}(E_b/N_0)$. The shape of the curve is *waterfall like* with a theoretical BEP curve to the left of curves for real-world systems. A system with impairments in the end-to-end transmission path, including the demodulator (think symbol classifier), increase the BEP for a given operating scenario. In a Wireless LAN or cable modem, for example, a low BEP is required to insure reliable information exchange. A large M is used here to indicate a large number of bits per second, per Hz of bandwidth. To see a typical BEP curve jump forward to Figure 10.

Pulse Shaping Filter Considerations

The pulse shape used for this matched filter mismatch study is the discrete-time version of the square-root raised-cosine pulse shape:

$$p_{\mathrm{SRC}}(t) = \begin{cases} 1-\alpha+4\alpha/\pi, & t=0 \\ \frac{\alpha}{\sqrt{2}}\Big[\big(1+\frac{2}{\pi}\big)\sin\big(\frac{\pi}{4\alpha}\big) & \\ \quad \big(1-\frac{2}{\pi}\big)\cos\big(\frac{\pi}{4\alpha}\big)\Big], & t=\pm\frac{T}{4\alpha} \\ \Big\{\sin\big[\pi t(1-\alpha)/T\big]+ & \\ 4\alpha t\cos\big[\pi t(1+\alpha)/T\big]/T\Big\}/ & \\ \Big\{\pi t\big[1-(4\alpha t/T)^2\big]/T\Big\}^{-1}, & \text{otherwise} \end{cases} \tag{1}$$

where T is the symbol period. The name used here is square-root raised cosine (SRC). The transmitted signal bandwidth when using SRC shaping is approximately $(1+\alpha)R_s$, where $R_s = R_b/\log_2 M$ is the symbol rate and R_b is the serial bit rate. Note m-ary signaling and SRC pulse shaping together together serve to increase spectral

efficiency in all the applications mentioned in the introductory paragraph.

The upper plot of Figure 4 shows the right half of an SRC pulse shape for $\alpha = 0.5$ and 0.25. The lower plot shows the result of passing the transmit pulse through a matched and mismatched receiver filter. The point of the SRC-SRC cascade is to provide spectral efficiency and insure that the pulse zero crossing occur at the adjacent symbol periods, i.e. zero ISI. For the mismatched case you can see ISI has crept in.

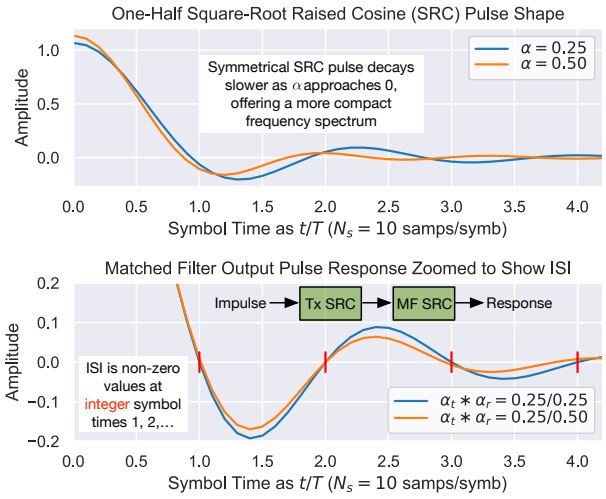


Fig. 4: *Plots of the SRC pulse shape (top) for $\alpha = 0.25$ and 0.5 and SRC-SRC cascading under a matched and mismatched receiver filter.*

For realizability considerations the discrete-time transmit pulse shaping filter and receiver matched filter are obtained by time shifting and truncating and then sampling by letting $t \to nT$. Residual errors at the matched filter output are present as a result of truncation as noted in both [Harris] and [Xing]. For small M values $\pm 6T$ is acceptable, but for the higher schemes considered in this paper we found increasing the filter length to $\pm 8T$ was required to avoid residual errors under matched pulse shape conditions. The residual errors at the zero crossings shown in the bottom half of Figure 4, but now for an ensemble transmit symbols, is shown in Figure 5. Here we see that the errors increase as α decreases.

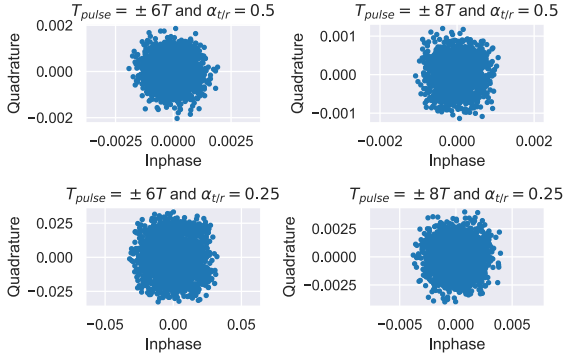


Fig. 5: *Matched SRC filters at transmit and receiver showing residual error due to FIR filter truncation of the doubly infinite pulse response [Rappaport], for a nominal maximum eye opening of ± 1.*

Semi-Analytic Bit Error Probability

Semi-analytic BEP (SA-BEP) calculation allows for fast and efficient analysis when a linear channel exists from the AWGN noise injection point to the receiver detector [Tranter]. A block diagram, which applies to the matched filter mismatch scenario of this paper, is shown in Figure 6. The variable z_k is the complex baseband detector decision statistic, as the receiver matched filter is sampled at the symbol rate, $R_s = 1/T$, nominally at the maximum eye opening. ISI is present in z_k due to pulse shape mismatch and other impairments such as timing error, static phase error, and even phase jitter. This corresponds to an ensemble of conditional Gaussian probabilities. The variance σ_w^2, for each the real/imaginary parts (inphase/quadrature), is calculated using

$$\sigma_w^2 = N_0 \cdot \sum_{n=0}^{N_{\text{taps}}-1} |p_r[n]|^2, \tag{2}$$

where the variance of the additive white Gaussian noise is denoted N_0 and $p_r[n]$ is the matched filter impulse response consisting of N_{taps}. The value of σ_w found in the conditional error probability of the following subsections is a function of N_0, which is set to give the desired average received energy per symbol E_s (note the energy per bit E_b is $E_s/\log_2(M)$) to noise power spectral density ratio, i.e., E_s/N_0 or E_b/N_0. This allows full BEP curves to be generated using just a single ensemble of ISI patterns. The calculation of N_0, taking into account the fact that the total noise power is split between real/imaginary (or in digital communications theory notation inphase/quadrature) parts is given by

$$N_0 = \frac{E_s}{2 \cdot 10^{(E_s/N_0)_{\text{dB}}/10}} \tag{3}$$

To be clear, $(E_s/N_0)_{\text{dB}}$ is the desired receiver operating point. In the software simulation model we set $(E_b/N_0)_{\text{dB}}$, convert to $(E_s/N_0)_{\text{dB}}$, arrive at N_0 for a fixed E_s, then finally determine σ_w. Note the 2 in the denominator of (3) serves to split the total noise power between the in-phase and quadrature components.

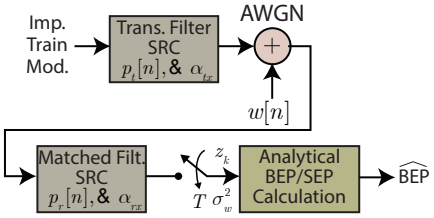


Fig. 6: *Block diagram describing how for a linear channel from the noise injection point to the detector, enables the use of semi-analytic BEP calculation over a more time consuming Monte-Carlo simulation.*

The SA-BEP method first calculates the symbol error probability by averaging over the ensemble of conditional Gaussian probabilities

$$P_{E,\text{symb}} = \frac{1}{N} \sum_{k=1}^{N} \Pr\{\text{Error}|z_k, \sigma_w, \text{other impairments}\} \tag{4}$$

where N is the number of symbols simulated to create the ensemble. For the m-ary schemes MPSK and MQAM we further assume that Gray coding (adjacent symbols differ by only one bit) is employed [Ziemer], and the BEP values of interest are small,

allowing the bit error probability to be directly obtained from the symbol error probability via

$$\text{BEP} = \frac{P_{E,\text{symb}}}{\log_2(M)} \tag{5}$$

The *other impairments* noted in (4) refers to the fact that SA-BEP can also be used to model carrier phase error or symbol timing error.

For the SA-BEP analysis model what remains is to find expressions for the conditional error probabilities in (4). A feature in the analysis of both MPSK and MQAM, is that both schemes reside in a two dimensional signal space and we can freely translate and scale signal points to a *normalized location* to make the error probability equations easier to work with.

M-ary PSK

For MPSK with $M > 2$ the optimum decision region for symbol detection is a wedge shaped region having interior angle π/M, as shown in the right side of Figure 7.

A simple upper bound, accurate for our purposes, is described in [Ziemer] and [Craig], considers the perpendicular distance between the nominal signal space point following the matched filter and the wedge shaped decision boundary as shown in Figure 7.

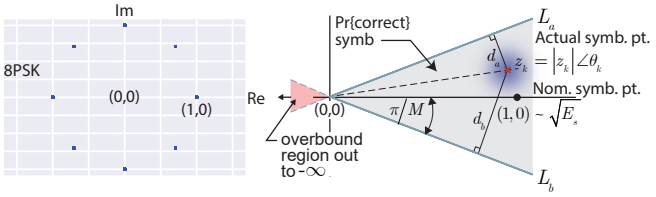


Fig. 7: *Formulation of the conditional symbol error probability of MPSK ($M = 8$ illustrated) given decision variable z_k.*

For unimpaired MPSK (no noise), we consider a normalized MPSK signal point, z_k, at angle zero to be the complex value $(1,0)$. Since z_k is actually a complex baseband signal sample, it can be viewed as the point $z_k = 1 + j0$ in the complex plane. The signal point length being one corresponds to setting $z_k = \sqrt{E_s} = 1$, where E_s is the symbol energy. The symbol error probability $P_{E,\text{symb}}$ is over bounded by the probability of lying above line L_a or below line L_b, when circularly symmetric Gaussian noise is now added to z_k. For the special case of $z_k = 1$ the probabilities of being above and below the lines are equal, hence this upper bound approximation results in

$$P_{E,\text{symb}} \simeq 2Q\left(\frac{z_k \cdot \sin(\pi/M)}{\sigma_w}\right) = 2Q\left(\frac{\sin(\pi/M)}{\sigma_w}\right), \tag{6}$$

where $Q(x)$ is the Gaussian Q function given by

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-t^2/2}\, dt. \tag{7}$$

Since we have assumed that $z_k = 1$ we use σ_w via N_0 to control the operating point, E_s/N_0, and hence also E_b/N_0. The over bound region, shown in light red in Figure 7, is due to double counting the error probability in this region.

With the bound only small differences are noted for the $M = 4$ case, and then only at very low E_b/N_0 values. The bound becomes tighter as M increases and as E_b/N_0 increases. We conclude that the bounding expression for $P_{E,\text{symb}}$ is adequate for use in semi-analytic BEP calculations at P_E values below 10^{-3}.

When matched filter mismatch is present the complex decision variable z_k, obtained by sampling the matched filter output, no longer sits at a normalized value of $(1,0) = 1\angle 0$. The scenario of a perturbed z_k is the real intent of Figure 7, where it shows two perpendicular distances, d_a and d_b, for an arbitrary z_k. We now use these distances to form the conditional probability of symbol error, and hence the Gray coded BEP. Using simple geometry to write d_a and d_b in terms of the angle π/M and $z_k = |z_k|e^{j\theta_k}$ we can finally write the conditional symbol error probability as

$$\begin{aligned} P_{E,\text{symb}}(z_k, \sigma_w) \;=\; & Q\left(\frac{|z_k|\sin(\pi/M - |\theta_k|)}{\sigma_w}\right) + \\ & Q\left(\frac{|z_k|\sin(\pi/M + |\theta_k|)}{\sigma_w}\right). \end{aligned} \tag{8}$$

M-ary Quadrature Amplitude Modulation

For MQAM the noise-free received symbols are scaled and translated to lie nominally at $(0,0)$ in the complex plane. Here we pattern the development of the SEP expression after [Ziemer]. The decision region for correct symbol detection detection is one of three types: (1) interior square, (2) left/right or top/bottom channel to infinity, (3) corners upper right/left and bottom right/left with two infinite sides, as depicted in Figure 8.

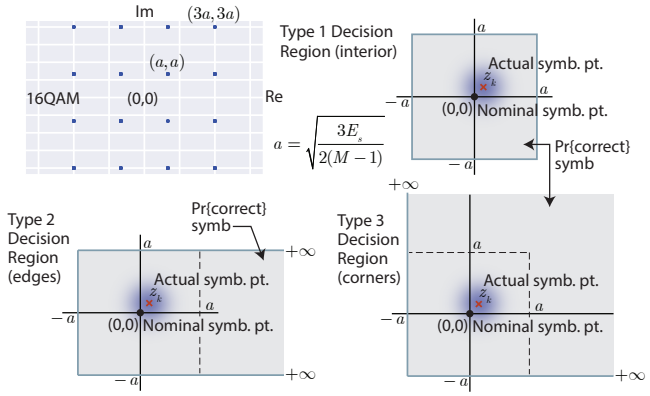


Fig. 8: *Formulation of the conditional symbol error probability of MQAM given decision variable z_k.*

Using simplifications similar to the MPSK case, we have the following equations for calculating the conditional SEP for symbol Types 1, 2, and 3. In the semi-analytic simulation software the symbol is known a priori, so in forming the average of (4) we choose the appropriate expression. For type 1 we have:

$$\begin{aligned} & P_{E|\text{type 1}}(z_k, \sigma_w|\text{type 1}) = \\ & \quad Q\left(\frac{a - \text{Re}\{z_k\}}{\sigma_w}\right) + Q\left(\frac{a + \text{Re}\{z_k\}}{\sigma_w}\right) \\ & \quad + Q\left(\frac{a - \text{Im}\{z_k\}}{\sigma_w}\right) + Q\left(\frac{a + \text{Im}\{z_k\}}{\sigma_w}\right) \end{aligned} \tag{9}$$

For type 2 we have:

$$\begin{aligned} & P_{E|\text{type 2}}(z_k, \sigma_w|\text{type 2}) = \\ & \quad Q\left(\frac{a - \text{Re}\{z_k\}}{\sigma_w}\right) + Q\left(\frac{a + \text{Re}\{z_k\}}{\sigma_w}\right) \\ & \quad + Q\left(\frac{a \pm \text{Im}\{z_k\}}{\sigma_w}\right) \end{aligned} \tag{10}$$

Finally for type 3 we have:

$$P_{E|\text{type 3}}(z_k, \sigma_w|\text{type 3}) = Q\left(\frac{a \pm \text{Re}\{z_k\}}{\sigma_w}\right) + Q\left(\frac{a \pm \text{Im}\{z_k\}}{\sigma_w}\right) \quad (11)$$

In all three conditional probability of bit error expressions, (9), (10), and (11), the variable a is defined is defined in terms of the energy per symbol, E_s and modulation order M using

$$a = \sqrt{\frac{3E_s}{2(M-1)}}. \quad (12)$$

Software Tools and Reproducible Science

All of the analysis and simulation software developed for this study is written in Python. It makes use of the *scipy-stack* and the authors GitHub project *scikit-dsp-comm* [Wickert1]. Two classes `MPSK_sa_BEP` and `MQAM_sa_BEP` do all of the heavy lifting. The code base specifics for this paper can be found on GitHub at [Wickert2]. The contents include Jupyter notebooks and code modules. All of this is open-source and freely available.

Results

In this section we consider the impact of filter mismatch in MPSK and MQAM. Equalization is not included in these first two studies. Next we consider how a short length equalizer can be employed to mitigate the mismatch performance losses, at increased system complexity.

Effects of Mismatch Filtering on MPSK

To limit the amount of data presented to the reader the figures shown for MPSK have a constant $\alpha_{\text{tx}} = .25$ while varying $\alpha_{\text{rx}} = .3$, .4, and .5. Later we provide heatmaps of E_b/N_0 degradation results over a range of α_{tx} and α_{rx} scenarios. Figure 9 shows IQ diagrams across orders of M while varying α_{rx}. The IQ diagrams plot the received symbols of the ideal matched filter system overlaid with the received symbols of a mismatched filter system. The left column shows that a small mismatch results in minimal error with every symbol being clearly defined, even at 32PSK. However, on the far right we see a more extreme case of mismatch filtering resulting in more ISI. With less separation between symbols it is expected that higher orders of M are more affected by mismatch filtering.

Figure 10 shows a row of BEP curves for $M = 16$ while varying α_{rx}. The BEP Curves show how mismatch filtering affects P_E across E_b/N_0 while comparing it to a theory curve. Each curve plots the theory curve for the modulation type, a SA-BEP curve with a perfect matched filter, and a SA-BEP Curve that varies α_{rx} with a constant α_{tx}. These results correspond to the first row of IQ diagrams presented in Figure 9. On the left we see a small mismatch results in minimal error with all three curves tightly together. On the right we a large degradation, denoted as the increase in E_b/N_0 to achieve the same P_E with perfect matched filter.

Figure 11 shows one row of eye diagrams across for $M = 8$ while varying α_{rx}. The eye diagrams show the effects of the added ISI introduced by mismatched filtering at the maximum eye opening sampling instant of the symbols. The same pattern of Figures 9 and 10 are seen here in terms of eye diagrams: a wide eye on the left side at the sampling instance meaning less ISI and noise. While on the right side the ISI begins to close the eye. Not shown here, higher orders of M are more perturbed by the introduction of mismatch filtering.

Figure 12 shows the degradation over various BEP threshold values of $\{10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}\}$, $M = 4$, 8, 16, and 32, and many combinations of $\alpha_{\text{tx}}/\alpha_{\text{rx}} \in [1/2, 2]$. The degradation is the measured shift in E_b/N_0 in dB between ideal theory and a system with filter mismatch at a particular BEP threshold. As M increases and $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ moves above or below 1 the degradation gets worse. With the worse degradation happening at $M = 32$ and $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ reaching the extremes of 1.2 and 2. Note degradation values of less than 0.01 dB are considered insignificant and are entered in the heatmap as zero values.

Effects of Mismatch Filtering on MQAM

Here we show only IQ diagrams for $\alpha_{\text{tx}} = .25$ while varying $\alpha_{\text{rx}} = .3$, .4, and .5. As in the MPSK case later we provide E_b/N_0 degradation results over a range of α_{tx} and α_{rx} values. Figure 9 shows two rows of IQ diagrams for $M = 16, 256$ while varying α_{rx}. The IQ diagrams plot the received symbols of the ideal matched filter system overlaid with the received symbols of a mismatched filter system. The left column shows that a small mismatch results in minimal error with every symbol being clearly defined, even at 256QAM. However, on the far right we see a more extreme case of mismatch filtering resulting in serious ISI, particularly for 256QAM. With less separation between symbols we expected large E_b/N_0 degradation will occur in the BEP plots.

Figure 14 repeats Figure 12 for MQAM. Results are similar for low modulation M, but the degradation for 256QAM is more serious than 32MPSK. This is not surprising when one considers the IQ diagrams, i.e., signal points are closer in MQAM than MPSK.

With Constrained Use of Equalization

The above results for MPSK and MQAM show that the ISI introduced from mismatch filtering is the greatest at highest modulation orders of, i.e., M, i.e., 32PSK and 256QAM, and when $\alpha_{\text{tx}} = .25$ and $\alpha_{\text{rx}} = .5$. In this subsection we briefly show how even a very simple adaptive equalizer can mitigate filter mismatch. An 11-tap equalizer is chosen to jointly minimize mismatch ISI yet balance noise enhancement. The short tap design can adapt quickly and minimize system complexity. To fit the SA-BEP analysis framework the equalizer is designed for fixed operation at $E_b/N_0 = 20$ dB, while the SA-BEP simulation is run for 20 dB $\leq (E_b/N_0)_{\text{dB}} \leq$ 25dB. In general an equalizer for digital communications is made adaptive using the least mean-square (LMS) adaptation algorithm [Ziemer] to minimize the mean-square error (MMSE) between the filter output and hard decision symbol estimates.

Figure 15 shows the effects of mismatch filtering when paired with a short length equalizer on 256QAM and $\alpha_{\text{tx}}/\alpha_{\text{rx}} = .25/.5$. The E_b/N_0 degradation is brought to about 1 dB at $P_E = 10^{-6}$. As you can see from Figure 15 the equalizer drastically reduces the ISI introduced by the filter mismatch. Even though the equalizer is designed for an operating point of 20dB it performs well across the entire range of E_b/N_0.

Concluding Discussion and Future Work

The effects of mismatch filtering on lower orders of M in both MPSK and MQAM, in particular $M = 4$, are almost negligible. With greater than .1dB E_b/N_0 degradation when the $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ ratio reaching the extremes of 1/2 and 2. The effects of mismatch

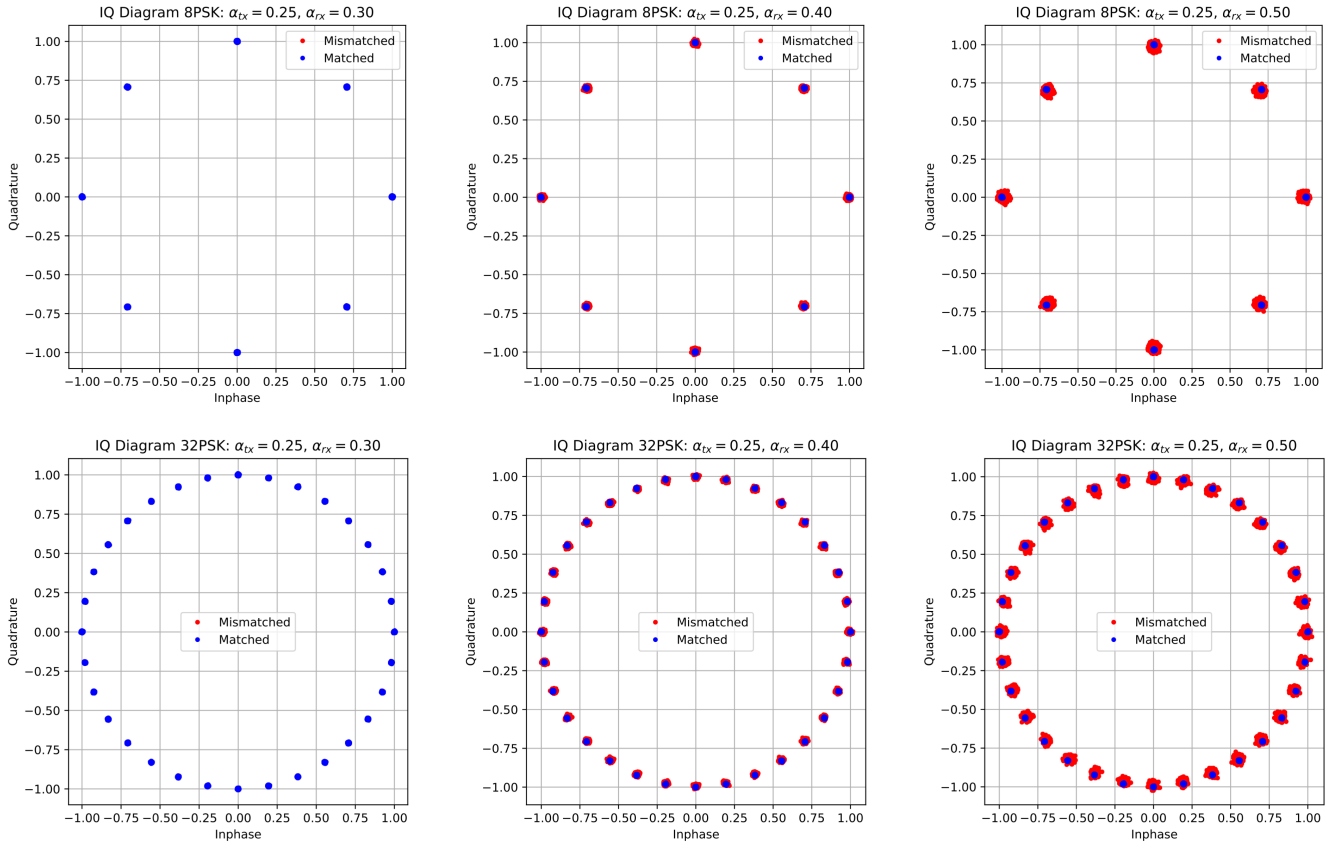


Fig. 9: *Two rows of IQ Diagrams showing the effects of mismatch filtering; The order of M increases with row number, $M = 8, 32$; $\alpha_{tx} = .25$ is fixed across all columns, while α_{rx} increases with column number as .3, .4, .5.*

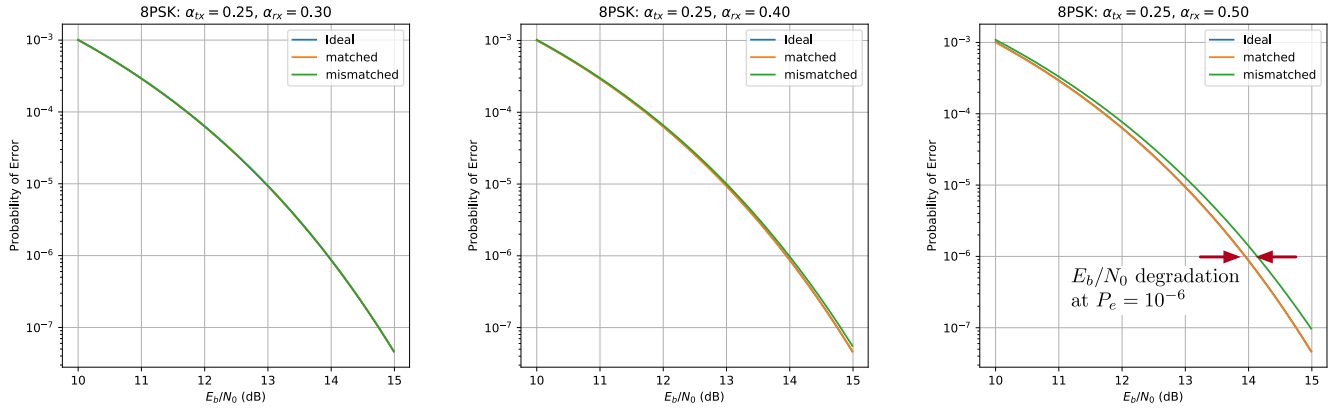


Fig. 10: *One row of BEP Curves showing the effects of mismatch filtering; Here M is fixed at 16; $\alpha_{tx} = .25$ across the columns, while α_{rx} increases with column number as excess bandwidth factors of .3, .4, .5.*

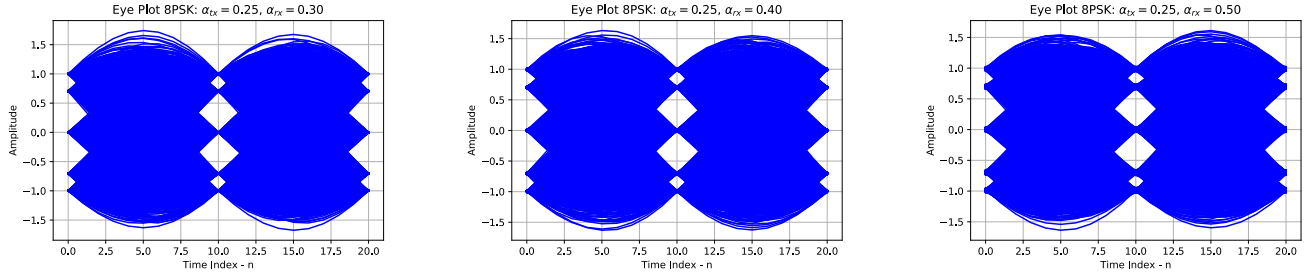


Fig. 11: *One row of of eye diagrams showing the effects of mismatch filtering; here M is fixed at 8; $\alpha_{tx} = .25$ across the columns, while α_{rx} increases with column number as excess bandwidth factors of .3, .4, .5.*

	α_{tx}	0.25	0.25	0.25	0.25	0.25	0.3	0.35	0.4	0.45	0.5
	α_{rx}	0.3	0.35	0.4	0.45	0.5	0.25	0.25	0.25	0.25	0.25
M	**BEP**	**E_b/N_0 Degradation (dB)**									
4	10^{-5}	0*	0*	1.00e-2	2.41e-2	4.37e-2	0*	0*	0*	2.40e-2	4.43e-2
4	10^{-6}	0*	0*	1.26e-2	3.01e-2	5.46e-2	0*	0*	1.26e-2	3.01e-2	5.52e-2
4	10^{-7}	0*	0*	1.53e-2	3.62e-2	6.56e-2	0*	0*	1.53e-2	3.61e-2	6.62e-2
4	10^{-8}	0*	0*	1.80e-2	4.23e-2	7.66e-2	0*	0*	1.80e-2	4.22e-2	7.72e-2
4	10^{-9}	0*	0*	2.06e-2	4.84e-2	8.77e-2	0*	0*	2.06e-2	4.84e-2	8.83e-2
8	10^{-5}	0*	0*	3.47e-2	8.15e-2	1.49e-1	0*	0*	3.48e-2	8.16e-2	1.49e-1
8	10^{-6}	0*	1.22e-2	4.39e-2	1.02e-1	1.87e-1	0*	1.21e-2	4.39e-2	1.03e-1	1.87e-1
8	10^{-7}	0*	1.49e-2	5.31e-2	1.24e-1	2.25e-1	0*	1.49e-2	5.31e-2	1.24e-1	2.25e-1
8	10^{-8}	0*	1.77e-2	6.23e-2	1.45e-1	2.62e-1	0*	1.77e-2	6.24e-2	1.45e-1	2.62e-1
8	10^{-9}	0*	2.06e-2	7.16e-2	1.65e-1	3.00e-1	0*	2.05e-2	7.16e-2	1.66e-1	2.99e-1
16	10^{-5}	0*	3.87e-2	1.32e-1	3.06e-1	5.61e-1	0*	3.88e-2	1.32e-1	3.06e-1	5.61e-1
16	10^{-6}	0*	4.92e-2	1.67e-1	3.86e-1	7.05e-1	0*	4.92e-2	1.67e-1	3.86e-1	7.05e-1
16	10^{-7}	1.13e-2	5.97e-2	2.02e-1	4.64e-1	8.46e-1	1.14e-2	5.98e-2	2.02e-1	4.64e-1	8.46e-1
16	10^{-8}	1.36e-2	7.03e-2	2.36e-1	5.42e-1	9.83e-1	1.36e-2	7.04e-2	2.36e-1	5.42e-1	9.83e-1
16	10^{-9}	1.58e-2	8.09e-2	2.71e-1	6.18e-1	1.11e+0	1.58e-2	8.10e-2	2.71e-1	6.18e-1	1.11e+0
32	10^{-5}	2.89e-2	1.46e-1	5.06e-1	1.22e+0	2.38e+0	2.90e-2	1.46e-1	5.06e-1	1.22e+0	2.38E+0
32	10^{-6}	3.72e-2	1.86e-1	6.43e-1	1.55e+0	3.04e+0	3.73e-2	1.86e-1	6.43e-1	1.55e+0	3.04E+0
32	10^{-7}	4.56e-2	2.26e-1	7.80e-1	1.87e+0	3.65e+0	4.56e-2	2.26e-1	7.80e-1	1.87e+0	3.64E+0
32	10^{-8}	5.40e-2	2.67e-1	9.14e-1	2.18e+0	4.17e+0	5.40e-2	2.67e-1	9.14e-1	2.18e+0	4.17E+0
32	10^{-9}	6.24e-2	3.07e-1	1.04e+0	2.46e+0	4.61e+0	6.25e-2	3.07e-1	1.04e+0	2.46e+0	4.61E+0

* degradation less than 0.01 dB; Tx/Rx Pulse Shape Span = ± 8 symbols

Fig. 12: *MPSK degradation resulting from filter mismatch.*

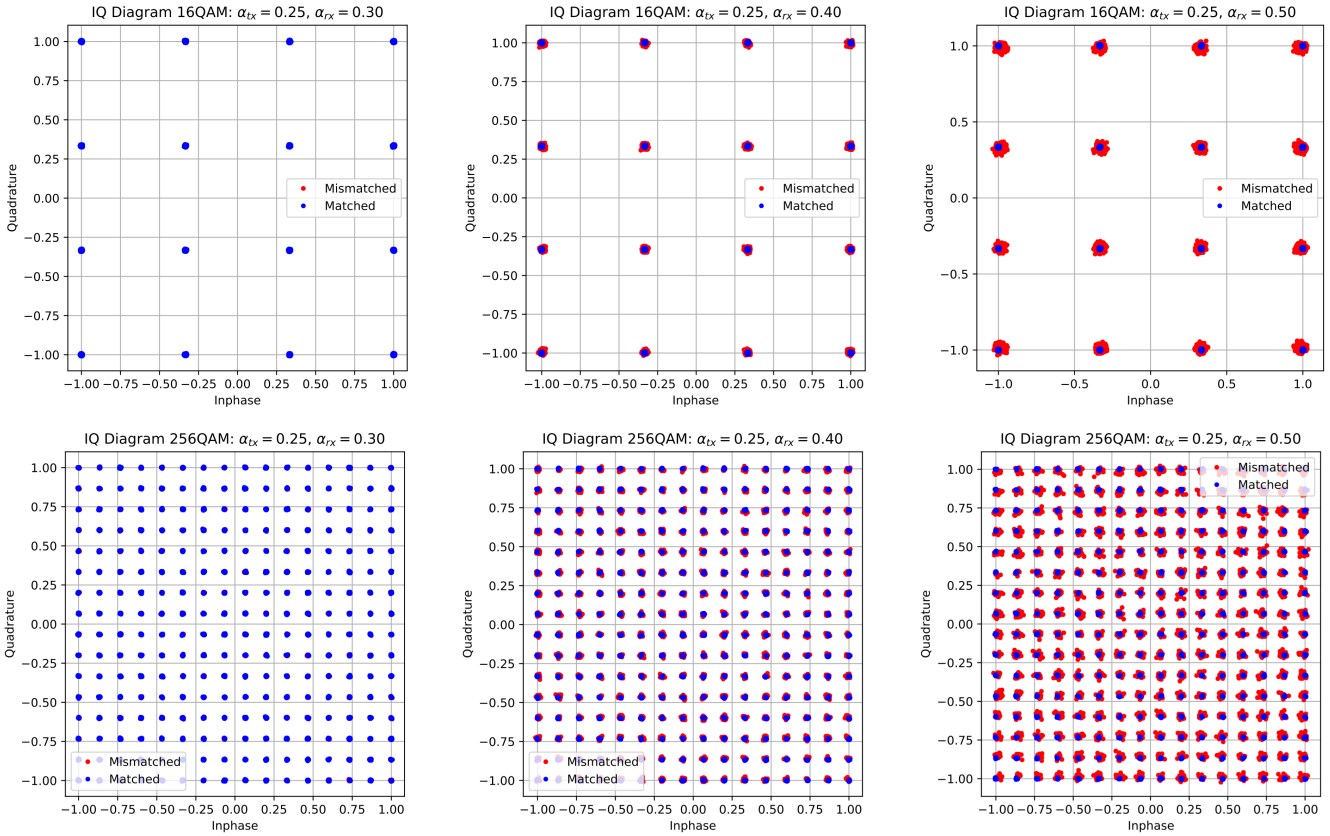


Fig. 13: *Two rows of IQ Diagrams showing the effects of mismatch filtering; The order of M increases with row number, $M = 16, 256$; $\alpha_{tx} = .25$ fixed across all columns, while α_{rx} increases with column number as .3, .4, .5.*

	α_{tx}	0.25	0.25	0.25	0.25	0.25	0.3	0.35	0.4	0.45	0.5
	α_{rx}	0.3	0.35	0.4	0.45	0.5	0.25	0.25	0.25	0.25	0.25
M	**BEP**	**E_b/N_0 Degradation (dB)**									
4	10^{-5}	0*	0*	0*	2.40e-2	4.42e-2	0*	0*	0*	2.40e-2	4.42e-2
4	10^{-6}	0*	0*	1.26e-2	3.00e-2	5.50e-2	0*	0*	1.26e-2	3.00e-2	5.50e-2
4	10^{-7}	0*	0*	1.53e-2	3.61e-2	6.59e-2	0*	0*	1.53e-2	3.60e-2	6.59e-2
4	10^{-8}	0*	0*	1.79e-2	4.21e-2	7.67e-2	0*	0*	1.79e-2	4.21e-2	7.67e-2
4	10^{-9}	0*	0*	2.06e-2	4.81e-2	8.75e-2	0*	0*	2.06e-2	4.81e-2	8.75e-2
16	10^{-5}	0*.	1.17e-2	4.79e-2	1.15e-1	2.11e-1	0*	1.17e-2	4.79e-2	1.15e-1	2.11e-1
16	10^{-6}	0*.	1.56e-2	6.08e-2	1.44e-1	2.65e-1	0*	1.56e-2	6.08e-2	1.44e-1	2.65e-1
16	10^{-7}	0*.	1.95e-2	7.37e-2	1.74e-1	3.18e-1	0*	1.95e-2	7.37e-2	1.74e-1	3.18e-1
16	10^{-8}	0*.	2.35e-2	8.67e-2	2.03e-1	3.71e-1	0*	2.35e-2	8.67e-2	2.03e-1	3.71e-1
16	10^{-9}	0*.	2.74e-2	9.97e-2	2.33e-1	4.23e-1	0*.	2.74e-2	9.97e-2	2.33e-1	4.23e-1
64	10^{-5}	3.80e-2	8.87e-2	2.40e-1	5.29e-1	9.67e-1	3.80e-2	8.87e-2	2.40e-1	5.29e-1	9.67e-1
64	10^{-6}	4.17e-2	1.05e-1	2.96e-1	6.60e-1	1.21e+0	4.16e-2	1.05e-1	2.96e-1	6.60e-1	1.21e+0
64	10^{-7}	4.53e-2	1.22e-1	3.51e-1	7.89e-1	1.46e+0	4.53e-2	1.22e-1	3.51e-1	7.89e-1	1.46e+0
64	10^{-8}	4.89e-2	1.39e-1	4.07e-1	9.16e-1	1.69e+0	4.89e-2	1.39e-1	4.07e-1	9.16e-1	1.69e+0
64	10^{-9}	5.25e-2	1.56e-1	4.61e-1	1.04e+0	1.92e+0	5.25e-2	1.56e-1	4.61e-1	1.04e+0	1.92e+0
256	10^{-5}	3.83e-2	2.44e-1	8.86e-1	2.27e+0	5.06e+0	3.85E-2	2.44e-1	8.86e-1	2.27e+0	5.06e+0
256	10^{-6}	5.23e-2	3.13e-1	1.13e+0	2.98e+0	7.24e+0	5.25E-2	3.13e-1	1.14e+0	2.98e+0	7.24e+0
256	10^{-7}	6.64e-2	3.83e-1	1.39e+0	3.72e+0	9.92e+0	6.66E-2	3.83e-1	1.39e+0	3.72e+0	9.96e+0
256	10^{-8}	8.06e-2	4.52e-1	1.64e+0	4.48e+0	1.17e+1	8.08E-2	4.53e-1	1.64e+0	4.48e+0	1.17e+1
256	10^{-9}	9.47e-2	5.22e-1	1.89e+0	5.18e+0	1.28e+1	9.50E-2	5.22e-1	1.89e+0	5.18e+0	1.28e+1

* degradation less than 0.01 dB; Tx/Rx Pulse Shape Span = ± 8 symbols

Fig. 14: *MQAM degradation resulting from filter mismatch.*

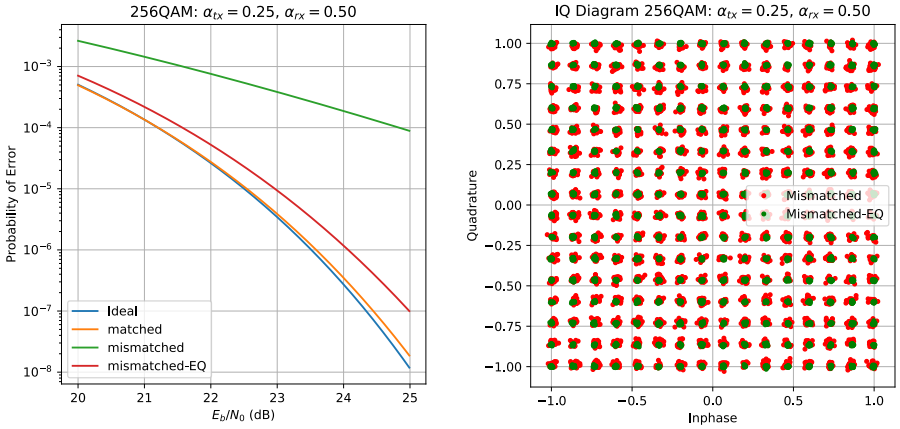


Fig. 15: *BEP Curve and IQ diagram showing the effects of mismatch filtering when using an 11-tap equalizer on 256QAM with $\alpha_{tx} = .25$ and $\alpha_{rx} = .5$; 11 taps offers a lot of improvement.*

filtering grow drastically as M increases and the BEP threshold point increases.

The IQ Diagrams show that the symbol clusters with mismatch are not circularly symmetric about the ideal symbol points. In general these *cluster clouds*, which we know result from ISI, appear biased toward the center of the IQ diagram. Characterizing the cluster cloud probability density function could serve as an alternative to SA-BEP technique presented in this paper.

Also observe is that the degradation values in the heatmaps are essentially symmetric for both MPSK and MQAM, with regard to the $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ ratio and its inverse. What this means is that the $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ ratio and its inverse give essentially the same E_b/N_0 dB degradation values. Does this make sense? The signal path is identical since the same two filters are connected in series (see Figure 6) in either case. Linear processing means the filter order can be reversed without changing the mismatch. What is different is that the white noise enters at the second filter, which is the receiver input. If the $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ ratio is less than one more WGN arrives at the receiver decision stage, but more signal energy also enters the receiver, in spite of being mismatched. If the $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ ratio is greater than one less AWGN arrives at the receiver decision stage, but less signal energy also enters the receiver, again in spite of being mismatched. Although a conjecture at the start of this research, the SA-BEP simulation results in Figures 12 and 14 support the above argument.

The use of SA-BEP modeling allowed this data to be quickly compiled and be easily repeatable. The code could quickly be modified to run any combination of MPSK, $\alpha_{\text{tx}}/\alpha_{\text{rx}}$ and present the data in any of the above formats. A purpose of this paper was reproducible science, for not only the author to be able to run the code but for any user to use the created code for their purposes and produce the same results. The use of SA-BEP modeling paired with the power and flexibility of object-oriented Python running in Jupyter notebooks accomplishes this goal.

REFERENCES

[Lindsey] W. Lindsey and M. Simon, Telecommunications Systems Engineering, original edition Prentice Hall, 1973. Reprint Dover Publications, 2011.

[Ziemer] R. Ziemer and W. Tranter, Principles of Communications, seventh edition, John Wiley, 2015.

[Proakis] G.J. Proakis, Digital Communications, 4th ed., McGraw Hill, 2001.

[Goldsmith] A. Goldsmith, Wireless Communications, Cambridge University Press, 2005.

[Tranter] W. Tranter, K. Shanmugan, T. Rappaport, and K. Kosbar, Principles of Communication Systems Simulation with Wireless Applications, Prentice Hall, 2004.

[Harris] F. Harris, C. Dick, S. Seshagiri, and K. Moerder, "An improved square-root nyquist shaping filter," Proceeding of the SDR 05 Technical Conference and Product Exposition, 2005.

[Xing] T. Xing, Y. Zhan, and J. Lu, "A Performance Optimized Design of Receiving Filter for Non-Ideally Shaped Modulated Signals," in *IEEE International Conference on Communications*, p. 914-919, 2008.

[Rappaport] T. Rappaport, Wireless Communications: Principles and Practice, Prentice Hall, 1999.

[Craig] J. Craig, "A New, Simple and Exact Result for Calculating the Probability of Error for Two-Dimensional Signal Constellations," in *IEEE Milcom '91*, p. 571-575, 1991.

[Wickert1] M. Wickert, "Scikit-dsp-comm: a collection of functions and classes to support signal processing and communications theory teaching and research," https://github.com/mwickert/scikit-dsp-comm.

[Wickert2] M. Wickert, "Matched filter mismatch losses: a Python sofware repository", https://github.com/mwickert/Matched_Filter_Mismatch_Losses.

Having your cake and eating it: Exploiting Python for programmer productivity and performance on micro-core architectures using ePython

Maurice Jamieson[‡*], Nick Brown[‡], Sihang Liu[‡]

Abstract—Micro-core architectures combine many simple, low memory, low power computing cores together in a single package. These can be used as a co-processor or standalone but due to limited on-chip memory and esoteric nature of the hardware, writing efficient parallel codes for these chips is challenging. In this paper we discuss our very low memory implementation of Python, ePython, supporting the rapid development of parallel Python codes for these co-processors. An offload abstraction is introduced, where programmers decorate specific functions in their Python code, running under any Python interpreter on the host CPU, with the underlying technology then taking care of the low level data movement, scheduling and ePython execution on the micro-core co-processor. A benchmark solving Laplace's equation for diffusion via Jacobi iteration is used to explore the performance of ePython on three different micro-core architectures, and introduces work around native compilation for micro-cores and the performance advantages that this can provide.

Index Terms—ePython, micro-cores, RISC-V, MicroBlaze, PicoRV32, Epiphany

Introduction

Micro-core architectures combine many simple, low power, cores on a single processor package. These micro-core architectures, providing significant parallelism and performance for low power but a major limitation is programmer productivity, where typically developers must write code in C, linked to low level libraries. Furthermore they must possess a deep understanding of the technology and address esoteric aspects including ensuring consistency with a (sometimes very) weak memory model, aligning data to word boundaries correctly, and the lack of basic features such as IO. As such, even the few experts who are able to program these chips struggle when it comes to obtaining good performance.

It was our hypothesis that Python can significantly help here and this is the reason why we developed ePython, an implementation of Python designed specially for micro-core architectures. Providing execution via both an interpreter (at around 24KB in size) and native compilation of code, ePython enables Python programmers to easily offload specific kernels in their code onto the micro-cores. This involves the seamless transfer of code and data to the device, as well as the copying back of results from the device. With very small memory spaces, of around between 32KB to 64KB per core, in order to support usable datasets then the underlying technology must also be capable of taking advantage of slower by larger external memories, and abstract the low-level details involved in moving data between these areas from the programmer.

In this paper we describe the use of ePython to program micro-cores from the perspective of supporting the offload of specific functions from an existing Python application, and then execute these kernels on the micro-cores. This technology currently supports a variety of micro-core architectures which include the Adapteva Epiphany, Xilinx MicroBlaze, and RISC-V PicoRV32, these three being the targets explored in this paper. The paper is organised as follows; in the next section we explore the background to micro-cores in more detail, some of the Python frameworks used to program accelerators and embedded technologies, and describe ePython. The section which follows introduces our abstractions for offloading kernels in application code which can be running via any Python interpreter on the host, onto the micro-cores and how these might be used most effectively. This is then followed by a description of the lower level details of ePython, discussing some of the architectural decisions that have been made in order to support easy porting between architectures and to fit into the limited memory spaces available. We then explore the performance of ePython on our three architectures of interest, initially focussing on the interpretation approach, which is currently most mature, before comparing and contrasting this against native code generation. Lastly we draw some conclusions and discuss further work.

Background and related work

There are numerous micro-core architectures including the PEZY-SC2 [pezy-sc] which powered the top Green 500 machine until it was decommissioned in March 2019, the Kalray Boston [kalray] and, the Celerity [ajayi2017celerity]. The work and experiments described in this paper focuses on three distinct types of micro-core; the Epiphany [epiphany], MicroBlaze [microblaze], and PicoRV32 [picorv32]. Developed by Adapteva and packaged as a single physical chip, the Epiphany is still arguably one of the most ubiquitous consumer-grade micro-cores, even thought it was developed a few years ago. On the Epiphany version 3 (Epiphany-III) each of these cores consists of a RISC-based

* Corresponding author: maurice.jamieson@ed.ac.uk
‡ EPCC at the University of Edinburgh

CPU, 32KB of high bandwidth on-core local memory (SRAM), two DMA engines and a network interface. The Epiphany is a physical chip, and whilst this is common place with consumer grade CPUs, it is expensive (approx. $1 million) to tape out a physical design. As such, soft-cores are also commonplace, where reconfigurable logic chips (such as FPGAs) are *configured to behave like* a specific CPU design. These is the case with the other two micro-cores that we target in this paper, the MicroBlaze and PicoRV32, and from the end programmer's perspective this chip looks exactly like a physical CPU. Crucially this approach is much cheaper than fabricating physical cores, although typically the reconfigurable nature of the fabric imposes a reduced clock frequency compared to a physical core. Irrespective of whether the chip is physical or soft, they contain many cores, each with very limited amounts of memory, and the reason for picking these specific three technologies here is both their ubiquity, and also representation of a wider class of micro-cores. The micro-core architecture is applicable to a wide range of problem domains, and performance levels close to 2 GFLOPs per core have been demonstrated [epiphany-specifications] in the field of signal processing on the Epiphany. A major advantage to this technology is around power efficiency, for instance even though it was designed in 2013, the 16 core Epiphany-III, draws a maximum of 2 Watts and delivers 16 GFLOPs/Watt which is very impressive even by today's standards.

In addition to the micro-core CPU, one also requires a board to mount this chip and connect it to the outside world. Adapteva, who developed the Epiphany, also manufactured the Parallella [parallella] which is a single board computer (SBC). The Parallella combines a host dual core ARM A9 CPU, with 1 GB of DRAM and the 16 core Epiphany-III. The theoretical off-chip bandwidth of the Epiphany is 600 MB/s, however in practice the maximum obtainable is around 150 MB/s. For our two soft-cores we use the same base-board, a Pynq-II SBC, and a Xilinx Zynx-7020 reconfigurable FPGA. The Zynq-7020 chip is especially interesting, as in a single physical package not only is there the reconfigurable fabric which we can use to represent our micro-cores of interest, but furthermore a dual core ARM A9 CPU which runs Linux. Therefore in a single chip we have the combination of a dual-core host CPU on the one-hand, and logic configured as multiple micro-core CPUs on the other. The board also contains 512 MB RAM with an off-chip bandwidth of 131.25 MB/s. This specific FPGA contains 53,200 programmable Look-Up Tables (LUTs), and around 627 KBs of block RAM (BRAM). In fact it is this BRAM, effectively the amount of local memory per core, which is the limiting factor here and we can fit a maximum of eight 64 KB MicroBlaze or PicoRV32 CPUs and supporting infrastructure onto the Zynq, which is the configuration used throughout this paper.

Whilst we have picked these micro-core technologies due to their availability and popularity, in our opinion the MicroBlaze and PicoRV32 are the more interesting targets. The MicroBlaze is developed by Xilinx, a large multi-national corporation who also develop the underlying FPGAs and there is significant commitment by Xilinx to the technology. On the other-hand, the PicoRV32 is an implementation of the RISC-V Instruction Set Architecture (ISA). RISC-V is an open standard ISA and, first introduced in 2010, one of the major reasons for its popularity has been the fact that it is provided under open source licenses that do not require fees. This means that anyone is free to download the specification and develop their own implementation of the ISA, which indeed the PicoRV32 project have done. Furthermore, because all these CPUs share the same ISA, then the software eco-system can often be trivially ported between CPUs. This includes complex tooling such as compilers, debuggers, and profilers, which in themselves require significant development effort. Enabling developers of a new RISC-V based CPU to take the existing RISC-V software eco-system, and run this with little or no modifications on their chip, significantly reduces the effort required in developing such new CPUs. With a large community, who are mixture of commercial and academic contributors, RISC-V is currently a very topical and active area of research and commercial exploitation.

Whilst we have aimed to provide the reader some glimpse into the richness and diversity that makes up this area of CPU architectures, there is one specific characteristic that they all share. Namely, irrespective of whether one's micro-core is a physical Epiphany or soft-core such as the MicroBlaze, the programming of these technologies is technically challenging. Based on the severe limitations of the hardware, it will be of no surprise to the reader that they run *bare metal* (i.e. without an OS), and whilst some approaches beyond using C with the low level hardware specific library, such as OpenCL [opencl] and OpenMP [openmp] have been developed, these are at different levels of maturity and still require the programmer to explicitly program the chip using C at a very low level. Indeed, Xilinx's Pynq-II board has been designed around ease of use, loading up a default configuration of three MicroBlaze cores, and presenting a Python interface via the Jupyter notebook. However, Python only runs on the host ARM CPU of the Pynq-II and the programmer must still write C code, albeit embedded within the Jupyter notebook, to execute directly on each MicroBlaze and interface with them appropriately using host side code.

This programmability challenge is made more severe when one considers the tiny amount of memory per core, for instance 32KB on the Epiphany and 64KB on the MicroBlaze and PicoRV32. Whilst a portion of the board's main DRAM memory is often directly addressable by the micro-cores, there is a significant performance penalty when going *off chip* and using this in comparison with the on-core RAM. Therefore to achieve reasonable performance programmers have to either keep their code and data within the limits of the on-core memory, or design their codes to explicitly cache and pre-fetch. Regardless, this adds considerable additional complexity to any non-trivial codes and, it is our firm belief that this should be abstracted by the programming technology. Potentially this is where the programmer productivity gains of Python can be of significant benefit to micro-cores, and it has already been seen that without an easy to use environment, then the adoption of this technology will be necessarily narrowed.

There are some other Python-based technologies in a somewhat similar space and arguably the most ubiquitous of these is MicroPython [micropython]. MicroPython is an implementation of Python for micro-controllers and is designed to be both lightweight and also to enable programmers to execute Python codes easily, as well as exploring the lower level details of the machines. Similarly to ePython, it can run bare metal on a variety of controllers or run on more mainstream machines such as Unix or Windows OSes. Whilst MicroPython is very interesting, it is fundamentally different from ePython in a number of respects. Firstly memory size, where MicroPython requires 256KB of code space and 16KB of RAM [micropython-website], and whilst this is small in comparison to more mainstream Python interpreters such as CPython, it is still significantly above the limitations of

micro-core architectures such as the Epiphany. In addition to the RAM, embedded controllers often contain dedicated Read Only Memory (ROM) too which can be flashed with the MicroPython code. This is the case with the pyboard, which is the official MicroPython microcontroller board, as it contains both 1MB of ROM and 192KB of RAM, and as such provides plenty of space for MicroPython. In contrast, micro-cores are CPUs and tend not to have such ROM associated with them, and therefore ePython has a much more limited memory space within which it can work. The ePython interpreter and runtime code size is 24KB on the Epiphany (compared against MicroPython's 256KB), and because it must fit into the very limited CPU's RAM, was architected from day one to achieve this by adopting specific design decisions. The other big difference between MicroPython and ePython is that of parallelism because, whilst there is multi-threading support in MicroPython, parallelism is not the first class concern of this technology and there is more limited support for writing parallel codes to run over a multiple cores concurrently. We had to provide this in ePython because the vast majority of micro-core architectures contain multiple cores that must interoperate.

Numba [numba] is an annotation driven approach to accelerating and offloading Python kernels, where the programmer decorates specific functions in their code and these will be compiled into native machine code for execution. For instance the *@jit* decorator indicates that a specific function should be just-in-time (JIT) compiled and the native code executed rather than the Python code. Their approach has been extended to GPUs, where functions can be decorated with *@cuda.jit* which will execute them on the GPU and perform all data movement necessary. The management of data on the device is also possible via in-built functions such as *cuda.to_device* to copy specific data to the GPU. The machine code for kernels that this technology generates is larger than the memory spaces available in micro-core architectures, so it is not applicable directly for our target architecture, however Numba's use of annotations is a very convenient way of marking which functions should be offloaded. Their approach is currently tightly coupled to GPUs, for instance when one launches a kernel they must explicitly specify some GPU specific concerns such as the number of GPU threads per block and number of blocks per grid, but the general idea of annotating functions in this manner could be applied more generally to micro-cores.

ePython

ePython, which was first introduced in [epython], is an implementation of a subset of Python for micro-core architectures and is designed to be portable across numerous technologies. The primary purpose of ePython was initially educational, and also as a research vehicle for understanding how best to program these architectures and prototyping applications upon them. ePython was initially created with the aim of allows a novice to go from *zero to hero*, i.e. with no prior experience write a simple parallel hello world example that runs on the micro-cores, in less than a minute. Due to the memory limitations of these architectures, the ePython virtual machine (which is the part that actually runs on the micro-core architectures) is around 24KB on the Epiphany, with the remaining 8KB of on-core memory used for user byte code, the stack, heap and communications. It is possible for byte code, the stack and heap to overflow into shared memory transparently, but there is a performance impact when doing so. ePython also supports a rich set of message passing primitives such as point to point messages, reductions and broadcasts between the cores, and it is also possible to run *virtual cores* where the host CPU behaves like micro-cores and can pass messages between themselves as normal. The code listing below illustrates a simple example which is executed directly on the micro-cores and launched from the host command line such as issuing *epython example.py*. In this example, each micro-core will generate a random integer between 0 and 100 and then perform a collective message passing reduction to determine the maximum random number (due to the *"max"* operator) which is then displayed by each core.

```
1 from parallel import reduce
2 from random import randint
3
4 a = reduce(randint(0,100), "max")
5 print "The highest random number is " + str(a)
```

This approach was initially developed with the objective of running rather simple examples on the micro-cores directly and exposing programmers to the fundamental ideas behind parallelism in a convenient programming language. As such, ePython implements a subset of Python 2.7, and was initially focussed around the imperative aspects of the code with features such as garbage collection, and has been extended to include other aspects of the Python language as time has progressed, although does not provide a complete implementation due to memory space limits. However, going beyond the work of [epython], we realised that there was potential for ePython to support real-world applications on micro-cores, but to do so a more powerful approach to programmer interaction was required. This is because not all parts of an application are necessarily suited for offloading to micro-cores, so an approach where specific functions can be selected for offload conveniently was required to extend the technology, which is the focus of the next section.

Offloading application kernels

We have extended ePython to couple it with existing Python codes running in any Python interpreter on the host CPU. As illustrated in Figure 1, ePython is comprised of three main components:

- A module which programmers import into their application Python code, running under any Python interpreter on the host, which provides abstractions and underlying support for handling the offloading of select code fragments to the micro-cores
- An ePython support host process which performs code preparation (such as lexing and parsing) as well as some general management functionality such as the marshalling and control of the micro-cores
- An execution engine on each of the micro-cores. This contains an architecture specific runtime, paired with either the ePython interpreter or execution of native code which has been generated from the programmer's offloaded Python kernels.

The first component is connected to the second via POSIX shared memory, and the method by which the second component connects to the third is architecturally specific depending upon the micro-cores in question. The targets considered in this paper all connect with the host via memory mapped regions, where specific portions of the memory space are visible to both host and micro-cores, although these tend to be mapped at different absolute addresses between the host and micro-core. The underlying mechanism for achieving this communication is abstracted as a set of services in the host's monitor, and the micro-core's architecture

specific runtime. Therefore we have been able to support ePython on other architectures which connect using different mechanisms, such as RS232 via a daughter board, by providing alternative implementations of the services.

In this section we explore the first of these components, and more specifically the abstractions provided which enable Python programmers to direct what aspects of their code should run on the micro-cores.

Similar to the approach taken by Numba, the programmer annotates kernel functions to be offloaded to the micro-cores with a specific decorator, *@offload*. When the CPU Python code executes a call to functions marked with this decorator it will, behind the scenes, run that function using ePython on the micro-cores, passing any input values and sending back return values. The code listing below provides an illustration of this, where the *mykernel* function has been marked with *@offload*, so the call to *mykernel* at line 7 will launch this kernel on each micro-core, passing the argument *22* to each function execution and obtain, as a list, the return value from the kernel (in this case the integer value *10* from each core). In this example the only modification required to standard Python code for offloading is importing the *epython* module and decorating the function. Function arguments are pass by reference, so it is only a reference to the data which is passed to the micro-cores upon kernel invocation, with ePython transparently transferring data as it is requiring during the execution of the kernel.

```
1 from epython import offload
2 @offload
3 def mykernel(a):
4   print "Hello with " + str(a)
5   return 10
6
7 print mykernel(22)
```

Behind the scenes to implement this offload functionality, upon initialisation the *epython* module will parse the full Python code and search for functions that might need to be executed on the micro-cores, such as the kernels and functions that they call into. These are extracted out into a separate Python file which is passed to ePython, which itself is then executed as a subprocess. Launched on each micro-core, low level message passing communications pass between the micro-cores and Python interpreter on the host via the ePython support host process. Upon the initialisation of a user's Python code on the CPU, the imported *epython* module interrogates ePython about the byte code location of all remotely executable functions, which is then stored. Subsequently, to execute a specific function on the micro-cores the host sends the stored byte code location of the function to the target core(s) in combination with an execution token. All output from the ePython subprocess is forwarded to standard output, so the programmer can still perform IO and view error messages raised by their offloaded kernels. If a programmer wishes to import specific modules in their kernels, then they can utilise either the *import* or *use* statements at the top of the function body.

Kernel execution options

The semantics of the offload is that, by default, the kernel will be executed on all available micro-cores and the caller will block until these have been executed. It is possible to override these defaults to further control the behaviour of kernel launch and execution. This is achieved by either providing explicit arguments to the decorator such as *@offload(async=True)* which will apply the option to all executions of the kernel, or alternatively the programmer can provide options as a named argument to the function call. An example of the later is *mykernel(22, async=True)*, which will override the arguments of the decorator for this specific kernel invocation. There are a number of possible options which can be used to control kernel behaviour:

Asynchronous execution
: By providing the argument *async=True* the execution of the kernel will proceed in a non-blocking manner where the function call will return a handler of type *KernelExecutionHandler* immediately. This object represents the state of the kernel execution over one or more micro-cores, and provides methods for testing kernel completion, waiting on kernel completion on all cores (and obtaining the results) and waiting for kernel completion on any core (and obtaining results.)

Auto
: The argument *auto=n*, where *n* is an integer representing the number of cores to execute the kernel over. This signifies that the programmer does not care which cores are used, but instead to run the kernel on *n* free micro-cores whenever these are available.

All
: The argument *all=True* will collectively execute the kernel on all available micro-cores.

Target
: The argument *target=n*, where *n* is either an integer core id or list of core ids, will guarantee to execute the kernel on those specific cores only. This can be useful if there is some distinct state or data held by core(s) which the programmer wants to utilise in their kernel.

Device
: The argument *device=d*, where *d* is the specifier of a type of micro-core architecture or a list of these and will execute the kernel on those types of specific micro-cores only. This is for programming heterogeneous micro-core systems which contain a number of micro-cores CPUs of different types, with device types defined for each available micro-core.

These options, specifically the placement options of *target*, *auto* and *all* can conflict if used together. Hence an order of precedence is defined and this is based upon the order in which they were introduced above. For instance if the programmer provides both *auto* and *target* then because *auto* has higher precedence it will be honoured and the *target* specifier ignored.

Scheduler

Using some of the options described previously can result in a situation where kernels are scheduled for execution, but the target cores are busy executing previous kernels. The *epython* module, imported by the entire Python application, implements a scheduler running inside a thread to handle this situation. The module keeps track of what cores are currently idle and which are active, as well as maintaining a list of outstanding kernel launches which are awaiting a free micro-core. Any kernel execution that can not be honoured is packaged up with additional information such as where to run the code and any arguments before being stored in a list. The scheduler will then scan through these waiting kernels and check whether the corresponding core can be used

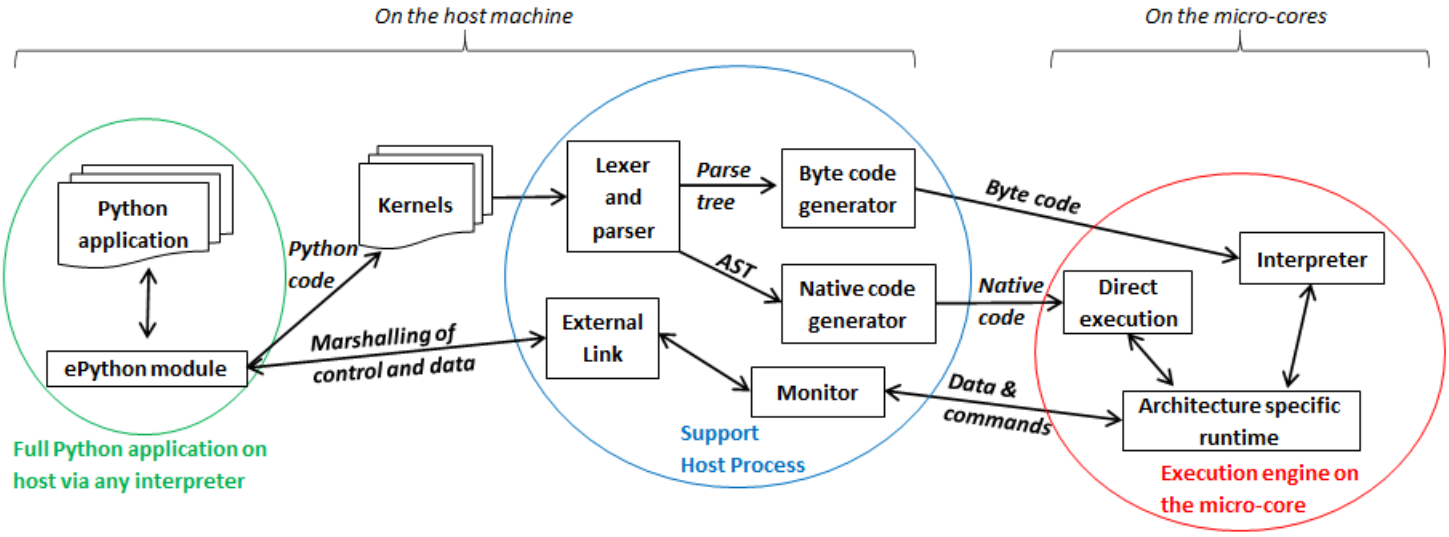


Fig. 1: *ePython architecture, connecting the programmer's Python code in any Python interpreter on the host, to execution on the micro-cores.*

to execute this kernel yet, and if so then the kernel is launched automatically. To ensure correctness a strict ordering, based upon the scheduling order, is maintained for kernel launches. Therefore, if kernel *A* is scheduled to run on core 0 and then kernel *B* is scheduled to run on the same core, ePython guarantees that *A* will execute on this core before *B*. Much of this is abstracted inside the *KernelExecutionHandler* class, object instances of which are returned as handlers from asynchronous kernel launches, and the class also contains methods for obtaining the general scheduling state such as how many kernel executions are currently running, and how many are scheduled and waiting to be run.

Working with arbitrarily large data-sets

It might seem apparent to the reader that one of the limitations of the approach thus described is the size of data that can be manipulated on the micro-cores. More specifically, very small data-sets can be copied into the micro-core local RAM which will provide optimal performance, but the majority of data sizes will instead need to be located in shared on-board but off-chip DRAM memory which is significantly slower. Using the abstractions described so far, the programmer would have to make a choice between the placement of their data and to manually copy in segments that they may wish to place in on-core memory for performance. The hierarchy of memories available to the micro-cores, and thus the Python programmer's kernels, is illustrated in Figure 2 for the Epiphany. From this diagram it can be seen that the problem is even more severe, as only a fraction of the host's 1GB DRAM is directly addressable by the micro-cores on the Epiphany (by default the shared segment is only 32MB in size). As such this significantly limits the data sizes that can be processed, as any data larger than this limit will not be able to reside in a location which is, by default, visible to the micro-cores.

This is in fact why the semantics of kernel arguments are pass by reference, rather than pass by value. Following a similar approach to CUDA's Unified Virtual Addressing (UVA) although, due to the simplicity of the micro-cores, achieving this entirely at the software level rather than hardware level, means that upon kernel invocation a simple reference is passed for each argument and it is this that the kernel works with. When the data is read from, or written to, by the micro-core then the ePython runtime

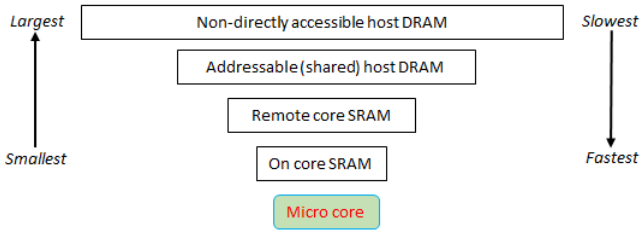


Fig. 2: *Illustration of memory hierarchy for the Epiphany.*

will, based upon this reference, perform the associated data movement operation with respect to the data's source location. Whilst it might appear that having to perform this data movement each time, potentially to or from a source location held far away in the memory hierarchy, is expensive, there are some further abstractions which can assist. Namely pre-fetching is supported which will utilise the micro-core's memory like a cache and copy in chunks ahead of time, then evicting them later on if necessary. On the Epiphany this is especially beneficial due to the two DMA engines per core, which can perform data transfers in a non-blocking manner and-so the cores can continue to work with data previously fetched whilst subsequent memory operations are in progress.

In combination with pass by reference and possible pre-fetching, it is also desirable for the programmer to be able to direct where in the memory hierarchy their data resides. This is supported via memory kinds. The code listing below illustrates a sketch of this, where the programmer uses the *memkind* class of the ePython module to allocate data. This enables them to direct where abouts in the memory hierarchy the data belongs and also the amount to allocate. Numerous memory kinds are provided and in this manner the programmer can easily direct what data belongs where, and then subsequently modify this if required without having to worry about any of the low-level nitty gritty details. It is still perfectly acceptable to declare variables normal Python style, without using memory kinds, and in such cases the variable belongs to the level of memory hierarchy that is currently in scope.

```
from epython import offload, memkind
import random

nums1 = memkind.Host(types.int, 1000)
nums2 = memkind.Host(types.int, 1000)

....

@offload
def mykernel(a, b):
  ....

print mykernel(nums1, nums2)
```

It is this same mechanism that enables device resident data, via the *Device* memory kind, to allocate the variables within the on-core memory of the micro-cores. ePython delegates to the memory kind the determination of the mapping between the requested index and the actual physical data region. Therefore, the memory kind can enable operations on memory spaces that are not directly visible to the micro-cores and, for instance, this is how we make visible the top level of the memory hierarchy of Figure 2 to the Epiphany and overcome the 32MB memory limit. In fact there is no inherent reason why the memory kinds must represent memory spaces at all, and in future could represent other facets including files or network connected resources.

Memory model

Python does not specify a standard memory model, with individual implementations being free to adopt whichever memory model they wish. In contrast to many other Python implementations, ePython adopts a rather weak memory model, which the programmer should be aware of.

Whenever a micro-core attempts to access a scalar variable or the index of an array, held elsewhere in the memory hierarchy, preference is given to any local copy held on that micro-core (cached). If there is no local copy, then a data transfer will be performed from where the data is physically located, effectively copying it to the micro-core and then caching it. The cache policy is write-through, where the locally held copy will then be used for all the reads, and writes are performed on both the local copy of data and also written back to the variable's location elsewhere in the hierarchy. Locally held cache copies of data are evicted automatically by the ePython runtime as required, such that the memory space can then be reused for subsequent data. Access to any data, whether it be a scalar or array element, held in memory locations outside the core will always first check whether there is a copy held locally, and if not perform the explicit data movement required. At the time of writing, by default the runtime waits until the data is required and then moves it, with the disadvantage of this approach is that it stalls execution until the memory operation completes. As such the programmer can, via decorating their code, instruct the data movement to be done ahead of time via non-blocking pre-fetching, thus not stalling the micro-cores on data access, and in the future this will likely become the default approach.

From the perspective of a single micro-core, updates to data are in-order and atomic. However between cores the model is weaker for performance reasons and to enable the reuse of data held locally rather than having to explicitly fetch it each time (for instance in situations where the same data element is used many times over by a kernel). This provides a simple and consistent model, and a big benefit within the context of simple micro-cores is that it requires limited support from the hardware and runtime software. However, the programmer should be aware of this because, if two or more kernels are working concurrently with the same data and both reading and writing to this, then ePython only imposes the atomicity of these updates. There is no guarantee around the order in which accesses from different cores will complete, or when kernels will see the data written by kernels on other cores. This is a somewhat different than that adopted by many multi-core CPUs, which are typically write-back and hence tend to only write data on cache flush, but do support a stronger memory model, often via directory based cache coherence.

ePython - a portable engine for parallel Python code execution

As illustrated in Figure 1, in addition to the *epython* module, there is also host side support code which runs as a separate process and an execution engine running on the target micro-cores. The later executes the programmer's code either via an interpreter or by natively compiling it. Both the ePython execution engine and and host-based support code are written in C and designed to be portable between architectures. Due to the very limited amount of memory available on these architectures, for the code running on the micro-cores it is not possible to link against the standard C library, or any other libraries for that matter. Instead, all the support functionality required, which in many cases is also architecture specific, is located in the ePython runtime. The idea is that the interpreter is entirely standard C99 code, and will call out to support functions in the runtime, thus meaning that to go from one architecture to another only a new runtime need be written. As such a version of the runtime must be provided for each architecture, and the API calls which must be implemented range from memory management and garbage collection, to communication between micro-cores and the host. The target architecture must provide at a minimum a C compiler, which itself is very common. We adopted this design as it provides both maximum portability and also considerable flexibility which is important for architecture specific optimisations.

When compiled the exact size of ePython depends upon the architecture being targeted. For instance with the Epiphany, where the ISA has been designed to result in small binaries, our compiled runtime is around 14KB and the interpreter 10KB. However on the PicoRV32 the binary size is around 40KB which is because the RISC-V ISA tends to result in more verbose machine code than the Epiphany's ISA. Furthermore, the Epiphany and MicroBlaze provide a Floating Point Unit (FPU) which supports (single precision) floating point arithmetic in hardware, whereas the PicoRV32 does not, and as such explicit floating point software support must also be included at the runtime level which increases the size of ePython. As the micro-cores are running bare-metal, ePython determines its own memory map, and whilst there is a standard ePython memory map that we defined in [epython], the exact location of where the separation between different memory areas lies, and the sizes of these areas, is flexible and abstracted by the architecture specific runtime. This is all abstracted by the runtime, and has no impact on the other parts of the code and therefore does not hinder portability.

The monitor of Figure 1 is directed by the micro-cores to perform certain activities, and runs via a thread on the host, polling for commands and data. It is through this mechanism that the micro-cores can *see* the programmer's host Python execution as an additional core, interacting with this via the sending or receiving of messages, which ultimately end up in the ePython module,

Description	Runtime (s)	Compared to Epiphany
Epiphany	18.20	N/a
MicroBlaze	129.08	7.1 times slower
PicoRV32	1014.96	55.76 times slower

TABLE 1: *Runtime of Jacobi benchmark on the three micro-core architectures using the ePython interpreter.*

and are used to marshall control and communicate data. These messages, instead of being sent to another micro-core, are sent to the monitor on the host which forwards them via POSIX shared memory to the host Python interpreter process. To achieve this, the same mechanism for passing messages between micro-cores can be used directly, without significant increases to size of ePython. The majority of support for marshalling control on the micro-cores is at the Python code level, where pre-written Python module code runs on the micro-cores to interpret the messages arriving from the host and then decoding these to determine which kernels to run or other actions to perform. This is important because, based upon the foundational concepts of message passing and task based parallelism, it meant that very limited modifications were required to the ePython execution engine on the micro-cores to support our offload approach, which is critical because memory is at so much of a premium.

Performance of the ePython interpreter

In this section we explore the performance of ePython on the three micro-core architectures that have been described in this paper, the Epiphany-III, the MicroBlaze, and PicoRV32. Due to the larger compiled size on the MicroBlaze and PicoRV32, in comparison to the Epiphany, these two architectures required 64KB of memory to run the full ePython stack. As discussed previously, the main limitation of the Zynq-7020 for hosting these soft-cores is the amount of memory available on the FPGA, and as such the maximum number of 64KB cores that can fit is eight. In order to provide a fair comparison, we also limit ourselves to eight Epiphany micro-cores in our experiments.

We chose a benchmark code for solving Laplace's equation for diffusion via Jacobi iteration. Jacobi iteration is a classic computational method for solving PDEs, and in this case we decompose our domain in one dimension across the micro-cores. Effectively in each iteration, every grid point is averaging across neighbouring values, and after each iteration a halo-swap is performed between pairs of micro-cores, to communicates the data on the exterior that is required for the next iteration. Furthermore, after each iteration the code calculates the relative residual, which is used to determine how far from the desired level of accuracy the current solution currently is. This involves each micro-core calculating its own local residual and then performing a reduction across the micro-cores to determine the overall global sum. All grid point numbers are single precision floating point, and we consider this benchmark interesting because it combines both floating point computation and communications. The runs described in this section are using the ePython interpreter, and Table 1 illustrates the runtime in seconds of each micro-core technology when our benchmark was executed upon it.

It can be seen in Table 1 that the Epiphany is by the far the most performant micro-core of the three that we are benchmarking in this section. This is potentially not surprising given the fact that it is a physical chip, and as such can run at a much higher clock frequency (600Mhz) compared to the two soft-cores (100Mhz). However, clearly from the results a six times difference in clock frequency is not the only reason for the performance gap, and other architectural differences play a role too. If we normalise for clock frequency, floating point operations on the PicoRV32 are still approximately 9 times slower than on the Epiphany, and this is because the Epiphany contains a hardware FPU which is superscalar, providing the capability of processing up to two floating point operations concurrently. By contrast, the PicoRV32 does not contain an FPU and as such all floating point arithmetic must be performed in software. Again normalising for clock frequency, array accesses are around 9.5 times slower on the PicoRV32 than on the Epiphany, and this is because on the Epiphany and MicroBlaze the cost of a memory load in cycle per instruction (CPI) is 1 cycle, whereas on the PicoRV32 it is 5 cycles. The Epiphany provides a variable length pipeline of up to eight stages and the MicroBlaze a five stage pipeline, by contrast the PicoRV32 is not pipelined and this results in an average CPI of 4 instructions, with the next instruction not being able to begin until the proceeding one has completed.

Cooking on gas - performance of native compilation

The performance limitations of the ePython interpreter become apparent when we compare against a version of the benchmark written in C and compiled on the host CPU. For instance, running on the Parallella's ARM Cortex-A9, a C version of the benchmark executes in 0.23 seconds which is around 80 times faster than the ePython version on eight cores of the Epiphany! This performance issue was one of the major facts that motivated us to explore native compilation of the programmers's Python code, such that it can execute directly on the micro-cores without the need for an interpreter. As per the architectural diagram of Figure 1, the natively compiled code can still take advantage of all the ePython runtime support, but crucially as both the runtime and the programmer's code are executed directly on bare metal, we believed that this would provide significant performance benefits. The ePython native code generator uses ahead-of-time (AOT) compilation, where the Python source code is compiled on the host machine to a native binary for execution on the micro-cores. Similarly to Micropython's Viper code emitter, the ePython native code generator uses machine word sizes (e.g. 32 bit on the Epiphany) and this is all transparent to the Python programmer, with their code matching the behaviour that would have been provided by the ePython interpreter. Like Micropython, but unlike Numba AOT compilation, the ePython code generation does not require the programmer to add type signatures to their offloaded kernels.

Unlike the Micropython just-in-time (JIT) and Numba compilers, the native code is not generated from existing Python bytecode, but instead from C source code generated from the abstract syntax tree (AST) created just after parsing and lexing the programmer's Python code. The resultant C source code is not a simple transliteration of Python to C, but instead the generation of optimal source code that supports the dynamic features of Python, whilst optimising memory access and arithmetic operations. We felt that this would be good approach because, unlike the bytecode-based approach, the ePython model is able to leverage the C compiler's extensive code optimisation routines at a higher level over a greater amount of source code, resulting in significantly faster code. To enable portability between architectures, the

Description	Runtime (s)
ePython native on Epiphany	0.031
C code on Epiphany	0.029
ePython native on AMD64 CPU	0.019
C code on AMD64 CPU	0.015

TABLE 2: *Runtime of natively compiled Python code via ePython, against bespoke C code, on both the Epiphany and AMD64 x86 CPU.*

generated C code is standard C99, and similarly to the interpreter calls into the runtime for anything which is architecturally specific.

Table 2 illustrates the runtime in seconds across different technologies when natively compiled. It can be seen that this is significantly faster, over 500 times, than using the ePython interpreter on the Epiphany. For comparison we have developed a C version of the benchmark specifically for the Epiphany and this represents the alternative of writing a bespoke implementation for the architecture. Developing such code in C is a significant undertaking, as the programmer must deal with numerous architecture specific complexities and low level concerns. Whilst it is this programming complexity that we believe Python has significant potential to overcome for micro-cores, we nevertheless felt it was interesting to include a C version as a comparison in a performance study such as this. We also ran a version of this benchmark on an AMD64 CPU (as both the ePython interpreter and native code generation support x86), which are ubiquitous in HPC and consumer grade computing.

This is currently the least mature part of ePython, and from Table 2 the reader can see that there is a small performance difference of around 10% on the Epiphany between ePython natively compiled code, and that written in C directly. The reason for this is the additional complexity that we have added into the natively compiled code to address the small memory spaces. We realised that a potential problem would be in natively compiling large Python kernels because it is very possible that these would result in an executable which is larger than the on-core memory or even the shared DRAM memory space. As such, the programmer's Python must be compiled in such a way that codes of an arbitrarily large size can be supported. Therefore, our approach adopts a dynamic loading approach, where a very small (approximately 1.5KB) bootloader is placed onto the micro-cores and this then *pulls in* the first function to execute. This bootloader intercepts all function calls, and upon a call it will check to see whether that function is currently held in on-core memory or not. If so then it will jump to that, or otherwise it will fetch the associated native code that comprises the function from the host, perform any required connections, and then execute it. Currently functions are flushed from the on-core memory upon completion of their execution, which is likely what accounts for the performance difference between the ePython native code and compiled C code, and in future this will be modified to be smarter, potentially with a garbage collection approach adopted instead.

Conclusions and further work

Micro-cores is a classification that covers a wide variety of processor technologies, and this is a thriving area which contains a number of vibrant communities. Whilst these are very interesting for a number of different reasons, a major challenge is around programmer productivity. We firmly believe that Python has a significant role to play here, but the peculiarities of the architectures, and more specifically the simplicity of the cores themselves and tiny amounts of associated memory, result in numerous challenges when looking to support a Python programming environment. As such, we initially realised that there is an important role for an implementation of Python which is very compact and can easily fit within the memory with space to spare for user code and data.

In this paper we have described ePython, an implementation of Python which is aimed to both support execution on micro-core CPUs, but also be highly portable between technologies. We have explored both the low-level aspects of how ePython is constructed, and also the abstractions provided to Python programmers such that they can easily offload specific parts of their application code onto the micro-cores. Being able to drive this offload by decorating functions within in existing applications is a very simple yet powerful way of interaction with the micro-cores, and the technology has also driven other aspects of the design, such as pass by reference.

The reader can clearly see that the performance obtained by ePython is very architecture specific, which is not surprising given the diversity of the different types of micro-cores and associated level of complexity. Whilst we expected a performance overhead associated with the ePython interpreter, the magnitude of this when compared to native code compilation surprised us. By contrast, one can see that the performance overhead of ePython can in large be ameliorated by using native code compilation to run the Python code directly on the micro-cores, without the need for an interpreter to be present. Therefore our present focus is in maturing the native code generation as we think this has demonstrated some worthwhile early results. In addition to exploring opportunities for further performance improvements, currently the architecture specific runtime library is not included in this dynamic loading, so the minimum code size is around 15KB (runtime and bootloader together). If we were to extend the dynamic loading approach to the runtime too, then the minimum size will be around 1.5KB plus the size of the largest function. This will open up the possibility of running over a number of additional micro-core architectures which contain tiny amounts of memory per core (only around 2KB or 3KB). Furthermore, our dynamic loading approach to native code compilation can be extended to fetch parts of third-party libraries, such as Numpy or Sklearn. This will require some thought, as we will need to split apart the ELF into its constituent components, but it would be of significant benefit to the micro-core software ecosystem if such a rich set of existing numerical frameworks could be supported by ePython.

ePython is currently focussed around version 2.7 of the language, and this reached end-of-life in January 2020. Therefore an important activity will be to upgrade ePython to support version 3 of the language, and we believe that the work done around the native code compilation is a key enabler. The reason for this is that implementing version 3 of the Python standard will require a number of extensions to the ePython interpreter which will push it beyond the current 24KB size. However this size issue is not present with the ePython native code compilation, not least because of our dynamic loading approach, and therefore it is our plan for the next ePython version to deprecate the interpreter and support Python version three based around native code compilation only.

REFERENCES

[picorv32] C. Wolf. *PicoRV32 - A Size-Optimized RISC-V CPU*, On Github, https://github.com/cliffordwolf/picorv32/, Last accessed June 2020

[pezy-sc] T. Ishii. *Introduction to PEZY-SC* http://accc.riken.jp/wp-content/uploads/2015/09/ishii.pdf, Last accessed June 2020

[kalray] B.D de Dinechin. *Kalray MPPA: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor* Hot Chips 27 Symposium (HCS), 2015 IEEE, pages 1--27

[ajayi2017celerity] S. Davidson et al. *Celerity: An Open-Source RISC-V Tiered Accelerator Fabric* IEEE Micro, Volume: 38, Issue: 2, March/April 2018, Pages 30 - 41

[epiphany] A. Olofsson. *Kickstarting high-performance energy-efficient manycore architectures with epiphany* 48th Asilomar Conference on Signals, Systems and Computers, 2014

[parallella] Adapteva. *Parallella-1.x Reference Manual* http://www.parallella.org/docs/parallella_manual.pdf, Rev 09, 2014

[microblaze] Xilinx. *MicroBlaze Processor Reference Guide* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf, 2018

[epiphany-specifications] Adapteva. *Epiphany Architecture Reference* http://www.adapteva.com/docs/epiphany_arch_ref.pdf, Rev 14, 2013

[opencl] J.E. Stone. D. Gohara. G. Shi. *OpenCL: A parallel programming standard for heterogeneous computing systems* Computing in science and engineering, Volume: 12, Issue: 3, May-June 2010, Pages 66 - 73

[openmp] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0* http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013

[micropython] D. P. George. *The MicroPython language* http://docs.micropython.org/en/latest/pyboard/reference/index.html, Last accessed June 2020

[micropython-website] MicroPython community. *MicroPython* https://micropython.org/, Last accessed June 2020

[numba] S.K. Lam. A. Pitrou. S. Seibert. *Numba: A LLVM-based Python JIT Compiler* Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015

[epython] N. Brown. *ePython: An Implementation of Python for the Many-core Epiphany Coprocessor* Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing, 2017

pandera: Statistical Data Validation of Pandas Dataframes

Niels Bantilan[‡§*]

https://youtu.be/PxTLD-ueNd4

Abstract—pandas is an essential tool in the data scientist's toolkit for modern data engineering, analysis, and modeling in the Python ecosystem. However, dataframes can often be difficult to reason about in terms of their data types and statistical properties as data is reshaped from its raw form to one that's ready for analysis. Here, I introduce pandera, an open source package that provides a flexible and expressive data validation API designed to make it easy for data wranglers to define dataframe schemas. These schemas execute logical and statistical assertions at runtime so that analysts can spend less time worrying about the correctness of their dataframes and more time obtaining insights and training models.

Index Terms—data validation, data engineering

Introduction

pandas [WM10] has become an indispensible part of the data scientist's tool chain, providing a powerful interface for data processing and analysis for tabular data. In recent years numerous open source projects have emerged to enhance and complement the core pandas API in various ways. For instance, pyjanitor [EJMZBSZZS19] [pyj], pandas-ply [pdpa], and siuba [sba] are projects that provide alternative data manipulation interfaces inspired by the R ecosystem, pandas-profiling [pdpb] automatically creates data visualizations and statistics of dataframes, and dask [Roc15] provides parallelization capabilities for a variety of data structures, pandas dataframes among them.

This paper introduces a data validation tool called pandera, which provides an intuitive, flexible, and expressive API for validating pandas data structures at runtime. The problems that this library attempts to address are two-fold. The first is that dataframes can be difficult to reason about in terms of their contents and properties, especially when they undergo many steps of transformations in complex data processing pipelines. The second is that, even though ensuring data quality is critical in many contexts like scientific reporting, data analytics, and machine learning, the data validation process can produce considerable cognitive and software development overhead. Therefore, this tool focuses on making it as easy as possible to perform data validation in a variety of contexts and workflows in order to lower the barrier to explicitly defining and enforcing the assumptions about data.

* *Corresponding author: niels.bantilan@gmail.com*
‡ *Talkspace*
§ *pyOpenSci*

In the following sections I outline the theoretical underpinnings and practical appications of data validation, describe in more detail the specific architecture and implementation of the pandera package, and compare and contrast it with similar tools in the Python and R ecosystems.

Data Validation Definition

Data validation is the process by which the data analyst decides whether or not a particular dataset fulfills certain properties that should hold true in order to be useful for some purpose, like modeling or visualization. In other words, data validation is a falsification process by which data is deemed valid with respect to a set of logical and statistical assumptions [VdLDJ18]. These assumptions are typically formed by interacting with the data, where the analyst may bring to bear some prior domain knowledge pertaining to the dataset and data manipulation task at hand. Notably, even with prior knowledge, exploratory data analysis is an essential part of the workflow that is part of the data wrangling process.

More formally, we can define data validation in its most simple form as a function:

$$v(x) \twoheadrightarrow \{True, False\} \tag{1}$$

Where v is the validation function, x is the data to validate, and the output is a boolean value. As [vdLdJ19] points out, the validation function v must be a surjective (onto) function that covers the function's entire range in order to be meaningful. To see why, consider a validation function that always returns *True* or always returns *False*. Such a function cannot falsify any instantiation of the dataset x and therefore fails to provide any meaningful information about the validity of any dataset[1]. Although the above formulation covers a wide variety of data structures, this paper will focus on tabular data.

Types of Validation Rules

[vdLdJ19] distinguishes between technical validation rules and domain-specific validation rules. Technical validation rules describe the variables, data types, and meta-properties of what constitutes a valid or invalid data structure, such as uniqueness and nullability. On the other hand, domain-specific validation rules

1. There are nuances around how to formulate the domain of the function v. For a more comprehensive formal treatment of data validation, refer to [vdLdJ19] and [VdLDJ18]

describe properties of the data that are specific to the particular topic under study. For example, a census dataset might contain `age`, `income`, `education`, and `job_category` columns that are encoded in specific ways depending on the way the census was conducted. Reasonable validation rules might be:

- The `age` and `income` variables must be positive integers.
- The `age` variable must be below 122[2].
- Records where `age` is below the legal working age should have `NA` values in the `income` field.
- `education` is an ordinal variable that must be a member of the ordered set `{none, high school, undergraduate, graduate}`.
- `job_category` is an unordered categorical variable that must be a member of the set `{professional, manegerial, service, clerical, agricultural, technical}`.

We can also reason about validation rules in terms of the statistical and distributional properties of the data under validation. We can think about at least two flavors of statistical validation rules: deterministic, and probabilistic. Probabilistic checks explicitly express uncertainty about the statistical property under test and encode notions of stochasticity and randomness. Conversely, deterministic checks express assertions about the data based on logical rules or functional dependencies that do not explicitly incorporate any assumptions about randomness into the validation function.

Often times we can express statistical properties about data using deterministic or probabilistic checks. For example, "the mean age among the `graduate` sample tends to be higher than that of the `undergraduate` sample in the surveyed population" can be verified deterministically by simply computing the means of the two samples and applying the logical rule $mean(age_{graduate}) > mean(age_{undergraduate})$. A probabilistic version of this check would be to perform a hypothesis test, like a t-test with a pre-defined alpha value. Most probabilistic checks can be reduced to deterministic checks, for instance by simply evaluating the truth/falseness of a validation rule using the test statistic that results from the hypothesis test and ignoring the p-value. Doing this simplifies the validation rule but trades off simplicity for being unable to express uncertainty and statistical significance. Other examples of such probabilistic checks might be:

- The `income` variable is positively correlated with the `education` variable.
- `income` is negatively correlated with the dummy variable `job_category_service`, which is a variable derived from the `job_category` column.

Data Validation in Practice

Data validation is part of a larger workflow that involves processing raw data to produce of some sort of statistical artifact like a model, visualization, or report. In principle, if one can write perfect, bug-free code that parses, cleans, and reshapes the data to produce these artifacts, data validation would not be necessary. In practice, however, data validation is critical for preventing the silent passing of an insidious class of data integrity error, which is otherwise difficult to catch without explicitly making assertions at runtime. These errors could lead to misleading visualizations, incorrect statistical inferences, and unexpected behavior in machine learning models. Explicit data validation becomes even more important when the end product artifacts inform business decisions, support scientific findings, or generate predictions about people or things in the real world.

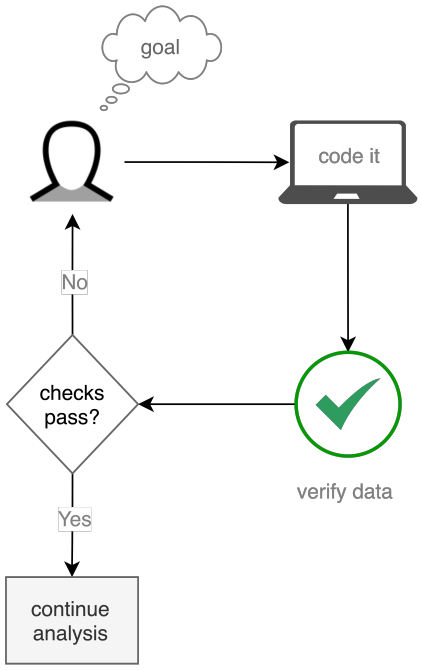


Fig. 1: *Data validation as an iterative software development process.*

Consider the process of constructing a dataset for training a machine learning model. In this context, the act of data validation is an iterative loop that begins with the analyst's objective and a mental model of what the data should "look" like. She then writes code to produce the dataset of interest, simultaneously inspecting, summarizing, and visualizing the data in an exploratory fashion, which in turn enables her to build some intuition and domain knowledge about the dataset.

She can then codify this intuition as a set of assumptions, implemented as a validation function, which can be called against the data to ensure that they adhere to those assumptions. If the validation function evaluates to `False` against the data during development time, the analyst must decide whether to refactor the processing logic to fulfill the validation rules or modify the rules themselves[3].

In addition to enforcing correctness at runtime, the resulting validation function also documents the current state of assumptions about the dataset for the benefit of future readers or maintainers of the codebase.

The role of the analyst, therefore, is to encode assumptions about data as a validation function and maintain that function as new datasets pass through the processing pipeline and the

2. The age of the oldest person: https://en.wikipedia.org/wiki/List_of_the_verified_oldest_people

3. In the latter scenario, the degenerate case is to remove the validation function altogether, which exposes the program to the risks associated with silently passing data integrity errors. Practically, it is up to the analyst to determine an appropriate level of strictness that catches cases that would produce invalid outputs.

definition of valid data evolves over time. One thing to note here is that using version control software like git [git] would keep track of the changes of the validation rules, enabling maintainers or readers of the codebase to inspect the evolution of the contract that the data must fulfill to be considered valid.

Design Principles

`pandera` is a flexible and expressive API for `pandas` data validation, where the goal is to provide a data engineering tool that (i) helps pandas users reason about what clean data means for their particular data processing task and (ii) enforce those assumptions at run-time. The following are the principles that have thus far guided the development of this project:

- Expressing validation rules should feel familiar to `pandas` users.
- Data validation should be compatible with the different workflows and tools in the data science toolbelt without a lot of setup or configuration.
- Defining custom validation rules should be easy.
- The validation interface should make the debugging process easier.
- Integration with existing code should be as seamless as possible.

These principles articulate the use cases that I had when surveying the Python ecosystem for `pandas` data validation tools.

Architecture

`pandera` helps users define schemas as contracts that a `pandas` dataframe must fulfill. This contract specifies deterministic and statistical properties that must hold true to be considered valid with respect to a particular analysis. Since `pandera` is primarily a data engineering tool, the validation function defined in Equation (1) needs to be slightly refactored:

$$s(v,x) \rightarrow \begin{cases} \text{x}, & \text{if } v(x) = true \\ \text{error}, & \text{otherwise} \end{cases} \tag{2}$$

Where s is a *schema* function that takes the validation function from Equation (1) and some data as input and returns the data itself if it is valid and an *error* otherwise. In `pandera`, the *error* is implemented as a `SchemaError` exception that contains the invalid data as well as a `pandas` dataframe of failure cases that contains the index and failure case values that caused the exception.

The primary rationale for extending validation functions in this way is that it enables users to compose schemas with data processing functions, for example, $s \circ f(x)$ is a composite function that first applies a data processing function f to the dataset x and then validates the output with the schema s. Another possible composite function, $f \circ s(x)$, applies the validation function to x before applying the f, effectively guaranteeing that inputs to f fulfill the contract enforced by s.

This formulation of data validation facilitates the interleaving of data processing and validation code in a flexible manner, allowing the user to decide the critical points of failure in a pipeline where data validation would make it more robust to abherrant data values.

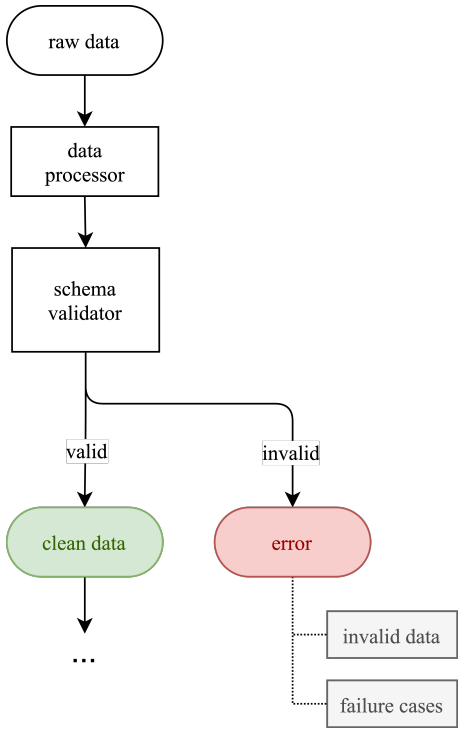


Fig. 2: *High-level architecture of `pandera`. In the simplest case, raw data passes through a data processor, is checked by a schema validator, and flows through to the next stage of the analysis pipeline if the validation checks pass, otherwise an error is raised.*

Core Features

DataFrameSchemas as Contracts

The main concepts of `pandera` are *schemas*, *schema components*, and *checks*. Schemas are callable objects that are initialized with validation rules. When called with compatible data as an input argument, a schema object returns the data itself if the validation checks pass and raises a `SchemaError` when they fail. Schema components behave in the same way as schemas but are primarily used to specify validation rules for specific parts of a `pandas` object, e.g. columns in a dataframe. Finally, checks allow the users to express validation rules in relation to the type of data that the schema or schema component are able to validate.

More specifically, the central objects in pandera are the `DataFrameSchema`, `Column`, and `Check`. Together, these objects enable users to express schemas upfront as contracts of logically grouped sets of validation rules that operate on pandas dataframes. For example, consider a simple dataset containing data about people, where each row is a person and each column is an attribute about that person:

```
import pandas as pd

dataframe = pd.DataFrame({
    "person_id": [1, 2, 3, 4],
    "height_in_feet": [6.5, 7, 6.1, 5.1],
    "date_of_birth": pd.to_datetime([
        "2005", "2000", "1995", "2000",
    ]),
    "education": [
        "highschool", "undergrad", "grad", "undergrad",
    ],
})
```

We can see from inspecting the column names and data values that we can bring some domain knowledge about the world to express

our assumptions about what are considered valid data.

```
import pandera as pa
from pandera import Column

typed_schema = pa.DataFrameSchema(
    {
        "person_id": Column(pa.Int),

        # numpy and pandas data type string
        # aliases are supported
        "height_in_feet": Column("float"),
        "date_of_birth": Column("datetime64[ns]"),

        # pandas dtypes are also supported
        # string dtype available in pandas v1.0.0+
        "education": Column(
            pd.StringDtype(),
            nullable=True
        ),
    },

    # coerce types when dataframe is validated
    coerce=True
)

typed_schema(dataframe)  # returns the dataframe
```

Validation Checks

The `typed_schema` above simply expresses the columns that are expected to be present in a valid dataframe and their associated data types. While this is useful, users can go further by making assertions about the data values that populate those columns:

```
import pandera as pa
from pandera import Column, Check

checked_schema = pa.DataFrameSchema(
    {
        "person_id": Column(
            pa.Int,
            Check.greater_than(0),
            allow_duplicates=False,
        ),
        "height_in_feet": Column(
            "float",
            Check.in_range(0, 10),
        ),
        "date_of_birth": Column(
           "datetime64[ns]",
           Check.less_than_or_equal_to(
               pd.Timestamp.now()
           ),
        ),
        "education": Column(
            pd.StringDtype(),
            Check.isin([
                "highschool",
                "undergrad",
                "grad",
            ]),
            nullable=True,
        ),
    },
    coerce=True
)
```

The schema definition above establishes the following properties about the data:

- the `person_id` column is a positive integer, which is a common way of encoding unique identifiers in a dataset. By setting `allow_duplicates` to `False`, the schema indicates that this column is a unique identifier in this dataset.
- `height_in_feet` is a positive float whose maximum value is 10 feet, which is a reasonable assumption for the maximum height of human beings.
- `date_of_birth` cannot be a date in the future.
- `education` can take on the acceptable values in the set `{"highschool", "undergrad", "grad"}`. Supposing that these data were collected in an online form where the `education` field input was optional, it would be appropriate to set `nullable` to `True` (this argument is `False` by default).

Error Reporting and Debugging

If a dataframe passed into the `schema` callable object does not pass the validation checks, `pandera` provides an informative error message:

```
invalid_dataframe = pd.DataFrame({
    "person_id": [6, 7, 8, 9],
    "height_in_feet": [-10, 20, 20, 5.1],
    "date_of_birth": pd.to_datetime([
        "2005", "2000", "1995", "2000",
    ]),
    "education": [
        "highschool", "undergrad", "grad", "undergrad",
    ],
})

checked_schema(invalid_dataframe)

# Exception raised:
SchemaError:
<Schema Column: 'height_in_feet' type=float>
failed element-wise validator 0:
<Check _in_range: in_range(0, 10)>
failure cases:
            index  count
failure_case
 20.0         [1, 2]      2
-10.0            [0]      1
```

The causes of the `SchemaError` are displayed as a dataframe where the `failure_case` index is the particular data value that failed the `Check.in_range` validation rule, the `index` column contains a list of index locations in the invalidated dataframe of the offending data values, and the `count` column summarizes the number of failure cases of that particular data value.

For finer-grained debugging, the analyst can catch the exception using the `try...except` pattern to access the data and failure cases as attributes in the `SchemaError` object:

```
from pandera.errors import SchemaError

try:
    checked_schema(invalid_dataframe)
except SchemaError as e:
    print("Failed check:", e.check)
    print("\nInvalidated dataframe:\n", e.data)
    print("\nFailure cases:\n", e.failure_cases)

# Output:
Failed check: <Check _in_range: in_range(0, 10)>

Invalidated dataframe:
   person_id  height_in_feet date_of_birth   education
0          6           -10.0    2005-01-01  highschool
1          7            20.0    2000-01-01   undergrad
2          8            20.0    1995-01-01        grad
3          9             5.1    2000-01-01        none

Failure cases:
   index  failure_case
0      0         -10.0
```

```
1      1          20.0
2      2          20.0
```

In this way, users can easily access and inspect the invalid dataframe and failure cases, which is especially useful in the context of long method chains of data transformations:

```
raw_data = ...  # get raw data
schema = ...  # define schema

try:
    clean_data = (
        raw_data
        .rename(...)
        .assign(...)
        .groupby(...)
        .apply(...)
        .pipe(schema)
    )
except SchemaError as e:
    # e.data will contain the resulting dataframe
    # from the groupby().apply() call.
    ...
```

Pipeline Integration

There are several ways to interleave `pandera` validation code with data processing code. As shown in the example above, one can use a schema by simply using it as a callable. Users can also sandwich data preprocessing code between two schemas; one schema that ensures the raw data fulfills certain assumptions, and another that ensures the processed data fulfills another set of assumptions that arise as a consequence of the data processing. The following code provides a toy example of this pattern:

```
in_schema = pa.DataFrameSchema({
    "x": Column(pa.Int)
})

out_schema = pa.DataFrameSchema({
    "x": Column(pa.Int),
    "x_doubled": Column(pa.Int),
    "x_squared": Column(pa.Int),
})

raw_data = pd.DataFrame({"x": [1, 2, 3]})
processed_data = (
    raw_data
    .pipe(in_schema)
    .assign(
        x_doubled=lambda d: d["x"] * 2,
        x_squared=lambda d: d["x"] ** 2,
    )
    .pipe(out_schema)
)
```

For more complex pipelines that handle multiple steps of data transformations with functions, `pandera` provides a decorator utility for validating the inputs and outputs of functions. The above example can be refactored into:

```
@pa.check_input(in_schema)
@pa.check_output(out_schema)
def process_data(raw_data):
    return raw_data.assign(
        x_doubled=lambda df: df["x"] * 2,
        x_squared=lambda df: df["x"] ** 2,
    )

processed_data = process_data(raw_data)
```

Custom Validation Rules

The `Check` class defines a suite of built-in methods for common operations, but expressing custom validation rules are easy. In the simplest case, a custom column check can be defined simply by passing a function into the `Check` constructor. This function needs to take as input a pandas `Series` and output either a boolean or a boolean `Series`, like so:

```
Column(checks=Check(lambda s: s.between(0, 1)))
```

The `element_wise` keyword argument changes the expected function signature to a single element in the column, for example, a logically equivalent implementation of the above validation rule would be:

```
Column(
    checks=Check(
        lambda x: 0 <= x <= 1, element_wise=True
    )
)
```

`Check` objects can also be used in the context of a `DataFrameSchema`, in which case the function argument should take as input a pandas `DataFrame` and output a boolean, a boolean `Series`, or a boolean `DataFrame`.

```
# assert that "col1" is greater than "col2"
schema = pa.DataFrameSchema(
    checks=Check(lambda df: df["col1"] > df["col2"])
)
```

Currently, in the case that the check function returns a boolean `Series` or `DataFrame`, all of the elements must be `True` in order for the validation check to pass.

Advanced Features

Hypothesis Testing

To provide a feature-complete data validation tool for data scientists, `pandera` subclasses the `Check` class to define the `Hypothesis` class for the purpose of expressing statistical hypothesis tests. To illustrate one of the use cases for this feature, consider a toy scientific study where a control group receives a placebo and a treatment group receives a drug that is hypothesized to improve physical endurance. The participants in this study then run on a treadmill (set at the same speed) for as long as they can, and running durations are collected for each individual.

Even before collecting the data, we can define a schema that expresses our expectations about a positive result:

```
from pandera import Hypothesis

endurance_study_schema = pa.DataFrameSchema({
    "subject_id": Column(pa.Int),
    "arm": Column(
        pa.String,
        Check.isin(["treatment", "control"])
    ),
    "duration": Column(
        pa.Float, checks=[
            Check.greater_than(0),
            Hypothesis.two_sample_ttest(
                # null hypothesis: the mean duration
                # of the treatment group is equal
                # to that of the control group.
                sample1="treatment",
                relationship="greater_than",
                sample2="control",
                groupby="arm",
                alpha=0.01,
            )
        ]
    )
})
```

Once the dataset is collected for this study, we can then pass it through the schema to validate the hypothesis that the group

receiving the drug increases physical endurance, as measured by running duration.

As of version `0.4.0`, the suite of built-in hypotheses is limited to the `two_sample_ttest` and `one_sample_ttest`, but creating custom hypotheses is straight-forward. To illustrate this, another common hypothesis test might be to check if a sample is normally distributed. Using the scipy.stats.normaltest function, one can write:

```
import numpy as np
from scipy import stats

dataframe = pd.DataFrame({
    "x1": np.random.normal(0, 1, size=1000),
})

schema = pa.DataFrameSchema({
    "x1": Column(
        checks=Hypothesis(
            test=stats.normaltest,
            # null hypothesis:
            # x1 is normally distributed with
            # alpha value of 0.01
            relationship=lambda k2, p: p > 0.01
        )
    ),
})

schema(dataframe)
```

Conditional Validation Rules

If we want to validate the values of one column conditioned on another, we can provide the other column name in the *groupby* argument. This changes the expected `Check` function signature to expect an input dictionary where the keys are discrete group levels in the conditional column and values are pandas `Series` objects containing subsets of the column of interest. Returning to the endurance study example, we could simply assert that the mean running duration of the treatment group is greater than that of the control group without assessing statistical significance:

```
simple_endurance_study_schema = pa.DataFrameSchema({
    "subject_id": Column(pa.Int),
    "arm": Column(
        pa.String,
        Check.isin(["treatment", "control"])
    ),
    "duration": Column(
        pa.Float, checks=[
            Check.greater_than(0),
            Check(
                lambda duration_by_arm: (
                    duration_by_arm["treatment"].mean()
                    > duration_by_arm["control"].mean()
                ),
                groupby="arm"
            )
        ]
    )
})
```

Functional dependencies are a type of conditional validation rule that expresses a constraint between two sets of variables in a relational data model [Arm74] [BFG+07]. For example, consider a dataset of biological species where each row is a species and each column is a classification in the classic hierarchy of `kingdom -> phylum -> class -> order ... -> species`. We can assert that "if two species are in the same `phylum`, then they must be in the same `kingdom`":

```
species_schema = pa.DataFrameSchema({
    "phylum": Column(pa.String),
    "kingdom": Column(
        pa.String,
        Check(
            # there exists only one unique kingdom
            # for species of the same phylum
            lambda kingdoms: all(
                kingdoms[phylum].nunique() == 1
                for phylum in kingdoms
            ),
            # this can also be a list of columns
            groupby="phylum"
        )
    )
})
```

However, in order to make the assertion "if two species are in the same `order`, then they must be in the same `class` and `phylum`", we have to use dataframe-level checks since the above pattern can only operate on values of a single column grouped by one or more columns.

```
species_schema = pa.DataFrameSchema(
    checks=Check(
        lambda df: (
            df.groupby("order")
            [["phylum", "class"]]
            .nunique() == 1
        )
    )
)
```

Use Case Vignettes

This section showcases the types of use cases that `pandera` is designed to address via hypothetical vignettes that nevertheless illustrate how `pandera` can be beneficial with respect to the maintainability and reproducibility of analysis/model pipeline code. These vignettes are based on my experience using this library in research and production contexts.

Catching Type Errors Early

Consider a dataset of records with the fields `age`, `occupation`, and `income`, where we would like to predict `income` as a function of the other variables. A common type error that arises, especially when processing unnormalized data or flat files, is the presence of values that violate our expectations based on domain knowledge about the world:

```
data = """age,occupation,income
30,nurse,90000
25,data_analyst,75000
45 years,mechanic,45000
21 year,community_organizer,41000
-100,wait_staff,27000
"""
```

In the above example, the `age` variable needs to be cleaned so that its values are positive integers, treating negative values as null.

```
import pandas as pd
import pandera as pa
from io import StringIO

schema = pa.DataFrameSchema(
    {
        "age": pa.Column(
            pa.Float,
            pa.Check.greater_than(0),
            nullable=True,
        ),
        "occupation": pa.Column(pa.String),
        "income": pa.Column(pa.Float),
    },
```

```
    coerce=True
)

pd.read_csv(StringIO(data)).pipe(schema)
# ValueError:
# invalid literal for int() with base 10: '45 years'
```

Defining a data cleaning function would be standard practice, but here we can augment this function with guard-rails that would catch `age` values that cannot be cast into a float type and convert negative values to nulls.

```
@pa.check_output(schema)
def clean_data(df):
    return df.assign(
        age=(
            df.age.str.replace("years?", "")
            .astype("float64").mask(lambda x: x < 0)
        )
    )

training_data = (
    pd.read_csv(StringIO(data)).pipe(clean_data)
)
```

The implementation of `clean_data` now needs to adhere to the `schema` defined above. Supposing that the data source is refreshed periodically from some raw data feed, additional records with age values like `22 years and 7 months` would be caught early in the data cleaning portion of the pipeline, and the implementation within `clean_data` would have to be refactored to normalize these kinds of more complicated values.

Though this may appear to be a trivial problem, validation rules on unstructured data types like text benefit greatly from even simple validation rules, like checking that values are non-empty strings and contain at least a minimum number of tokens, before sending the text through a tokenizer to produce a numerical vector representation of the text. Without these validation checks, these kinds of data integrity errors would pass silently through the pipeline, only to be unearthed after a potentially expensive model training run.

Reusable Schema Definitions

In contexts where the components of an ML pipeline are handled by different services, we can reuse and modify schemas for the purposes of model training and prediction. Since schemas are just python objects, schema definition code can be placed in a module e.g. `schemas.py`, which can then be imported by the model training and prediction modules.

```
# schemas.py
feature_schema = schema.remove_columns(["income"])
target_schema = pa.SeriesSchema(pa.Int, name="income")

# model_training.py
from schemas import feature_schema, target_schema

@pa.check_input(feature_schema, "features")
@pa.check_input(target_schema, "target")
def train_model(features, target):
    estimator = ...
    estimator.fit(features, target)
    return estimator

# model_prediction.py
from schemas import feature_schema, target_schema

@pa.check_input(feature_schema, "features")
@pa.check_output(target_schema)
def predict(estimator, features):
    predictions = estimator.predict(features)
    return pd.Series(predictions, name="income")
```

Unit Testing Statistically-Typed Functions

Once functions are decorated with `check_input` or `check_output`, we can write unit tests for them by generating synthetic data that produces the expected results. For example, here is a test example using `pytest` [pyt]:

```
# test_clean_data.py
import pandera as pa
import pytest

def test_clean_data():
    valid_data = pd.DataFrame({
        "age": ["20", "52", "33"],
        "occupation": ["barista", "doctor", "chef"],
        "income": [28000, 150000, 41000],
    })
    clean_data(valid_data)

    # non-normalized age raises an exception
    invalid_data = valid_data.copy()
    invalid_data.loc[0, "age"] = "20 years and 4 months"
    with pytest.raises(ValueError):
        clean_data(invalid_data)

    # income cannot be null
    invalid_null_income = valid_data.copy()
    invalid_null_income.loc[-1, "income"] = None
    with pytest.raises(pa.errors.SchemaError):
        clean_data(invalid_null_income)
```

This last use case would be further enhanced by property-based testing libraries like `hypothesis` [MHDC19] [MHDpb+20] that could be used to generate synthetic data against which to test schema-decorated functions.

Documentation

Documentation for `pandera` is hosted on ReadTheDocs, where tutorials on core and experimental features are available, in addition to full API documentation.

Limitations

The most notable limitation of `pandera` is the computational cost of running validation checks at runtime. This limitation applies to any data validation code, which trades off increased run-time for type safety and data integrity. The project currently uses `airspeed-velocity` [asv] for a few basic run-time and memory usage benchmarks, but more extensive performance profiling is warranted to give users a better sense of this trade-off. The other trade-off to consider is the additional development time associated with defining robust and meaningful schemas versus the time spent debugging silent data integrity issues, which is particularly costly in areas like machine learning where model debugging occurs after training a model.

A related limitation is that type-checking schemas are practical for large datasets (e.g. datasets that do not fit onto disk in a modern laptop), but validation checks that verify statistics on one or more columns can become expensive. For this reason, the default `Check` function signature is expected to be a `Series` in order to encourage users to use the optimized `pandas.Series` methods. In theory, `pandera` schemas can be coupled with parallelization tools like `dask` [Roc15] to perform data validation in these settings.

Two other limitations of the current state of the package are that:

- The built-in `Hypothesis` methods are currently limited in scope, and implementing wrapper methods to

the `scipy` implementations of commonly used distributional tests (e.g. normality test, chi-squared test, and KL-divergence) would encourage the use of hypothesis tests in schemas.
- Expressing functional dependencies is currently inelegant and would benefit from a higher-level abstraction to improve usability.

Roadmap

The `pandera` project started as a naive excursion into seeing whether pandas dataframes could be statically typed, as gradual typing is becoming adopted by the Python community since the `typing` module was introduced in Python 3.5. The project evolved into a tool that emphasizes the verification of the statistical properties of data, which requires run-time validation.

The direction of this project has been driven, in large part, by its contributors, and will continue to be via feature requests on the github repo. There are a number of experimental features that are currently available in version `0.4.0+` that aim to speed up the iteration loop of defining schemas at development time through interactive analysis:

- schema inference: the `pandera.infer_schema` function takes as input a dataframe and outputs an automatically generated draft schema that the user can iterate on.
- yaml/module serialization: this feature enables the user to write schemas (inferred or otherwise) to a yaml file or python script, which are editable artifacts to iterate on.

Additionally, a few feature proposals would benefit from discussion and feedback from the wider scientific computing and data science community:

- Synthetic data generation based on schema definitions [issue 200].
- Domain-specific schemas, types, and checks, e.g. for the machine learning use case, provide first-class support for validation checks between target and feature variables [issue 179].
- Expressing a tolerance level for the proportion of values that fail a validation `Check` [issue 183].

There are several ways to contribute for interested readers:

- Improving documentation by adding examples, fixing bugs, or clarifying the the writing.
- Feature requests: e.g. requests for additional built-in `Check` and `Hypotheses` methods.
- Submit new issues or pull requests for existing issues.

Related Tools

This project was inspired by the `schema` and `pandas_schema` Python packages and the `validate` R package [vdLdJ19]. Initially when assessing the Python landscape for `pandas`-centric data validation tools, I found that they did not match my use cases because they (a) often resulted in verbose and over-specified validation rulesets, (b) introduced many new library-specific concepts and configuration steps, (c) lacked documentation of core functionality and usage patterns, and/or (d) are no longer maintained.

Here is my assessment of data validation tools that are currently being maintained in the Python ecosystem:

- `great_expectations` [ge]: this is a mature, batteries-included data validation library centered around the concept of **expectations**. It provides a UI to manage validation rules and supports integrations with many database systems and data manipulation tools. This framework extends the `pandas.DataFrame` class to include validation methods prefixed with `expect_*` and a suite of built-in rules for common use cases. Defining custom validation rules involves subclassing the `PandasDataset` class and defining specially-decorated methods with function signatures that adhere to library-specific standards.
- `schema` [sch]: a light-weight data validator for generic Python data structures. This package and `pandera` share the schema interface where the schema object returns the data itself if valid and raises an `Exception` otherwise. However, this library does not provide additional functionality for `pandas` data structures.
- `pandas_schema` [ps]: a `pandas` data validation library with a comprehensive suite of built-in validators. This package was the inspiration for the *schema component* design where a `Column` object specifies properties of a dataframe column, albeit the specific implementations are considerably different. It provides built-in validators and supports defining custom validation rules. Unlike `pandera` which outputs the validated data, the output of validating a dataframe with `pandas_schema` is an iterable of errors that are intended to be inspected via `print` statements.

The key features that differentiate `pandera` from similar packages in the Python ecosystem are the following:

- `check_input` and `check_output` function decorators enable seamless integration with existing data processing/analysis code.
- `Check` validation rules are designed primarily for customizability, with built-in methods as a convenience for common validation rules.
- `Hypothesis` validation rules provide a tidy-first [W+14] interface for hypothesis testing.
- Ease of debugging, as `SchemaErrors` contain the invalidated data as well as a tidy dataframe of the failure cases with their corresponding column/index locations.
- Schema inference and serialization capabilities enable the creation of draft schemas that users can iterate on and refine.
- Clear and comprehensive documentation on core and advanced features.

Conclusion

This paper introduces the `pandera` package as a way of expressing assumptions about data and falsifying those assumptions at run time. This tool is geared toward helping data engineers and data scientists during the software development process, enabling them to make their data proprocessing workflows more readable, robust, and maintainable.

Acknowledgements

I would like to acknowledge the pyOpenSci community for their support and the `pandera` contributors who have made significant improvements and enhancements to the project.

REFERENCES

[Arm74] William Ward Armstrong. Dependency structures of data base relationships. In *IFIP congress*, volume 74, pages 580–583. Geneva, Switzerland, 1974.

[asv] airspeed velocity. Accessed: 29 May 2020. URL: https://github.com/airspeed-velocity/asv.

[BFG+07] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd international conference on data engineering*, pages 746–755. IEEE, 2007.

[EJMZBSZZS19] Eric J. Ma, Zachary Barry, Sam Zuckerman, and Zachary Sailer. pyjanitor: A Cleaner API for Cleaning Data. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 50 – 53, 2019. doi:10.25080/Majora-7ddc1dd1-007.

[ge] Great expectations: Always know what to expect from your data. Accessed: 29 May 2020. URL: https://github.com/great-expectations/great_expectations.

[git] Git. Accessed: 29 May 2020. URL: https://git-scm.com.

[MHDC19] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 11 2019. URL: http://dx.doi.org/10.21105/joss.01891, doi:10.21105/joss.01891.

[MHDpb+20] David R. MacIver, Zac Hatfield-Dodds, pyup.io bot, Alex Chan, Stuart Cook, Ryan Soklaski, David Chudzicki, jwg4, Alex Willmer, Tyler, Kyle Reeve, Grigorios Giannakopoulos, mulkieran, Emmanuel Leblond, Christopher Armstrong, Tyler Gibbons, Jeremy Thurgood, Paul Stiverson, SuperStormer, Alexander Shorin, David Mascharka, Peter C Kroon, Anne Archibald, Tom Prince, Mathieu PATUREL, dwest netflix, Tom McDermott, rdturnermtl, Graham Williamson, and Cory Benfield. HypothesisWorks/hypothesis: Hypothesis for Python - version 5.16.0, May 2020. URL: https://doi.org/10.5281/zenodo.3859851, doi:10.5281/zenodo.3859851.

[pdpa] pandas-ply. Accessed: 6 June 2020. URL: https://github.com/coursera/pandas-ply.

[pdpb] pandas-profiling. Accessed: 6 June 2020. URL: https://github.com/pandas-profiling/pandas-profiling.

[ps] Pandasschema. Accessed: 29 May 2020. URL: https://github.com/TMiguelT/PandasSchema.

[pyj] pyjanitor. Accessed: 6 June 2020. URL: https://github.com/ericmjl/pyjanitor.

[pyt] pytest. Accessed: 29 June 2020. URL: https://github.com/pytest-dev/pytest.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, number 130-136. Citeseer, 2015.

[sba] siuba. Accessed: 6 June 2020. URL: https://github.com/machow/siuba.

[sch] Schema: Schema validation just got pythonic. Accessed: 29 May 2020. URL: https://github.com/keleshev/schema.

[VdLDJ18] Mark Van der Loo and Edwin De Jonge. *Statistical data cleaning with applications in R*. Wiley Online Library, 2018.

[vdLdJ19] Mark PJ van der Loo and Edwin de Jonge. Data validation infrastructure for r. *arXiv preprint arXiv:1912.09759*, 2019.

[W+14] Hadley Wickham et al. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014.

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

Combining Physics-Based and Data-Driven Modeling for Pressure Prediction in Well Construction

Oney Erge[‡*], Eric van Oort[‡]

Abstract—A framework for combining physics-based and data-driven models to improve well construction is presented in this study. Additionally, the proposed approach provides a more robust and accurate model that mitigates the disadvantages of using purely physics-based or data-driven models. This approach can provide improved model-based control of drilling rig actuators (assocated with mud pumps, pipe handling systems, etc.).

Traditionally, models based on physics including Hagen-Poiseuille flow, Hooke's law, etc. are used during well construction. Physics-based models facilitate the design of the drilling plan and are vital to safely and successfully drilling wellbores. There are two major shortcomings, however, to using purely physics-based models. First, the models can be inaccurate if the physical dynamics are not fully accounted for. Accurately capturing data to describe these processes can be involved, complex or prohibitively expensive. Second, these models must be maintained and calibrated during drilling, which requires a large amount of operator input and is liable to human error. On the other hand, pure data-driven approaches are unable to represent underlying mechanism dynamics and often struggle to properly capture causal relationships. It is shown in this work combining physics and data-driven modeling provides a more robust framework for well planning and execution.

Machine learning techniques are combined with physics-based models via a rule-based stochastic hidden Markov model, using the modeling of frictional pressure losses during fluid circulation in the well as an example. Gaussian processes, neural networks and a deep learning model are trained and executed together with a physics model that is directly derived using first principles.

The results show that combination modeling can accurately predict the pressure losses even outperforming the physics-based and purely data-driven modeling. The proposed approach has a good potential to allow safer, optimized well construction operations.

Index Terms—deep learning, machine learning, combining physics-based modeling and data-driven modeling, hydraulics modeling, frictional pressure loss modeling.

1.Introduction

Well construction for energy (geothermal, oil and gas) is an inherently complex multi-disciplinary process. This process includes the interplay of solid, fluid and rock systems and requires managing subsurface events that are usually not measured or observed directly. In an attempt to digitally twin the process as much as possible, physics-based models are used during drilling wells. Physics-based models assist in the decision-making, especially in the well planning and design phases. Notably, modeling the fluid flow and managing the pressure in the wellbore is key to the safe and successful construction of wellbores.

In drilling, physics-based fluid flow models are derived directly from the first principles, using mass, momentum and energy equations. The sets of equations used to estimate the fluid behavior, are non-linear, lacking an analytical solution and can only be solved numerically. However, these numerical solutions are computationally expensive and generally fail to provide a practical real-time solution. Additionally, physics-based models have two major weaknesses: first, in a dynamic, complex system (i.e. well construction), all of the physics is typically not understood, given that the initial conditions are usually unknown. Therefore, the physics is not modeled accurately, resulting in inaccuracies; second, during any given process, input parameters change and the model should be constantly maintained to obtain useful results.

The advances of computational power and exponential growth of data have made data-driven modeling more viable and popular. Across various domains, data-driven models are being explored in an attempt to compare their performance to the physics-based models. The largest shortcoming of purely data-driven models, however, it that they are often black-box models, lacking a connection to underlying physics and thereby complicating interpretability. In other words, a specific outcome or decision by the data-driven models may not be logically understood by the human expert. Combining the two approaches may alleviate the crucial shortcoming of each approach individually.

In this work, we conducted physics-based simulations and trained data-driven models (neural networks and Gaussian processes) using an actual drilling dataset. We developed a hidden Markov model comprising of the process state and domain knowledge. The proposed approach shows potential to attain the best features of both approaches, and thereby allow for safer and more optimized well construction operations.

2.Literature Review

Combining physics-based and data-driven modeling, i.e. hybrid modeling, is a relatively new field of research [KWRK17]. The current literature is limited and spread across various domains. Consequently, the literature review below includes examples from distinct domains and diverse discussions. It should be noted that there are discrepancies in the terminology used in the literature of hybrid modeling. Although discussing the correct terminology/definitions is not within the scope of this paper, we do point out several of these discrepancies in the following.

[ACK13] investigated several prognostics techniques to predict the remaining useful life (RUL) of structural components (i.e.

* *Corresponding author: oneyerge@utexas.edu*
‡ *The University of Texas at Austin*

steel, aluminum), analyzing fatigue crack length growth over several loading cycles. They classified these prognostic techniques for condition-based maintenance (CBM) under three categories: physics-based, data-driven and hybrid models. In terms of data-driven modeling, the authors used neural networks (NN) and Gaussian Processes (GPs). For the physics-based and hybrid models, the authors referred to the two correlation type models as physics-based models and combined them with particle filters (PF) and Bayesian method (BM). They found that physics-based models provide significantly better accuracy at long-term RUL prediction. In case where the physics model is not available, purely data-driven models can be used for short term prediction.

[KWRK17] presented a framework to combine a physics-based model with neural networks, referring to the approach as physics guided neural networks (PGNN). They included a physics-based loss function in the learning objective of the neural networks, and applied this framework to model the temperature of lakes. By combining physics-based and data-driven modeling, better scientific consistency was achieved. They discovered the need for calibration to be a significant disadvantage of the physics-based models, which can be time-consuming. However, in this study, the physics-based model was actually a function containing a set of curve-fitted coefficients, i.e. a correlation. It is not derived using the first principles, but uses coefficients that are estimated using measurements taken on some physical dynamics. It is therefore not strictly a physics-based model. Their results showed that PGNN outperformed a purely data-driven method (NN) in terms of accuracy and consistency.

[KSB17] evaluated several approaches in the context of robots interacting with the physical world via analytical models, data-driven and hybrid models. They also analyzed the advantages and disadvantages of neural networks-based learning approaches for planar pushing. By applying neural networks to extract the physics model's inputs, they used the second stage of a reduced analytical model. In short, they used neural networks for perception and the analytical model for prediction. Two significant advantages of hybrid modeling were noted to be the reduction in the required amount of training data as well as the improvement in the generalization of physical interaction providing physically meaningful results.

[RRS+18] used data-driven modeling to accelerate the computational speed of a solver for incompressible flows. The computationally stiff part of the Poisson equation is solved through the data-driven approach, while the non-stiff part is handled with the incompressible flow solver. Orthogonal base functions are used in the reduced-order model space to solve the Poisson equation. By doing so, it is computationally significantly cheaper compared to a solver using finite differencing. Through data exchange between the full and reduced-order spaces, they achieved a significant reduction in the computational cost.

[KWOI18] noted that physics-based models, especially for drillstring dynamics, are not adequate for real-time operations. First, there are a lot of unknown parameters. Second, the physics model needs to be constantly tuned to fit the actual data. Their hybrid modeling approach was to use a recurrent neural network to train using the historical data of an ongoing drilling operation, and subsequently predict the drillstring dynamics in real-time. They recommended using the physics simulations of drillstring dynamics in case there is not enough data to properly train the network.

[DIX19] incorporated data-driven modeling into traditional turbulence modeling, with the intent to quantify and reduce uncertainties. They used statistical inference to extract model coefficients and discrepancies to improve the overall turbulent flow modeling accuracy. They combined physics-based and data-driven modeling in this order: first, the model discrepancy term is extracted via statistical inference from the datasets of interest. Then data-driven techniques are applied to calculate the discrepancies in the variables associated with the mean flow and turbulence. Finally, these discrepancies are given as input to Reynolds averaged Navier-Stokes (RANS) solvers as a correction to the traditional turbulence models in order to improve the overall accuracy. They highlighted that when using a data-driven approach, the uncertainties need to be presented and the physical and mathematical constraints need to be taken into account. Their work also showed that machine learning models need to be combined with physics models to produce credible results.

[GSK19] proposed a framework to combine physics-based (domain-focused) and data-driven (domain-agnostic) models to analyze physiological data and quantify the physiological state and abnormalities. They included expert knowledge into the modeling via a boosting-based ensemble learning algorithm, and presented several applications on how to combine various data sources to quantify neurological abnormalities. They used simulated data (heart rates simulated using differential equations) to compliment the accumulated measurements, and applied deep neural networks for predictions. At one of their examples using gait data, combining the domain and data-driven modeling allowed more accurate detection rates of abnormalities at a level of 40-50%. Overall, they strongly emphasized the use of simulated data to properly train the data-driven models by increasing the data quantity.

[PGAAEMH19] used a set of submodels in series to analyze the changes in the temperature and pressure across an engine system. They used a mix of physics-based (for charge-air intercooler, engine cylinder) and empirical (for intake and exhaust manifold) models to predict critical temperatures and pressures in the gas exchange system to facilitate model-based control. They also used artificial neural networks for the turbocharger submodel. By combining this set of submodels, the results showed a fair agreement with the measurements.

[MM19] outlined the advantages of combining physics-based and data-driven modeling to obtain improved inductive bias, improved scalability to larger datasets and better interpretability. They experimented using this approach with a system of pendulum, acrobat, cartpole and multibody dynamics. They proposed an Explicit Variational Gaussian Process, where they incorporated the domain knowledge through an explicit linear prior, which is developed using Newtonian mechanics. They concluded that black-box models ignore the structure of the problem and are less explainable, and increased interpretatbility by combining modeling approaches.

3. Modeling the Flow of Non-Newtonian Fluids in Well Construction

In the circulation system of a well under construction, the drilling fluid travels through the surface lines into the drillstring, passes through the nozzles of the bit and returns to the surface through the annulus. Frictional pressure losses in the circulation system are measured at the standpipe as standpipe pressure (SPP). Several parameters have a significant effect on the SPP, such as depth, flow rate, rotation of the drillstring, etc. The effects need to be

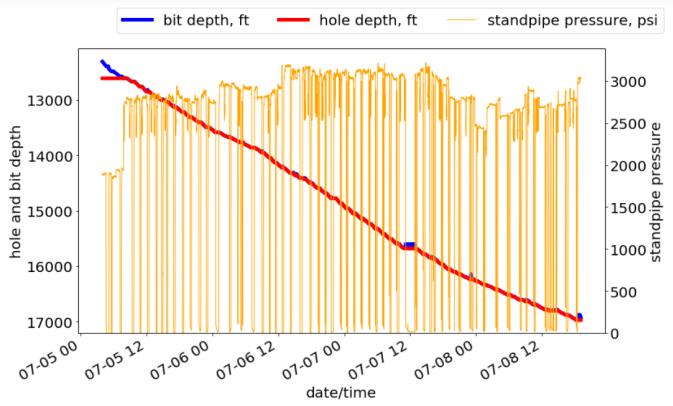


Fig. 1: *Hole and bit depth vs. time curve of Well A.*

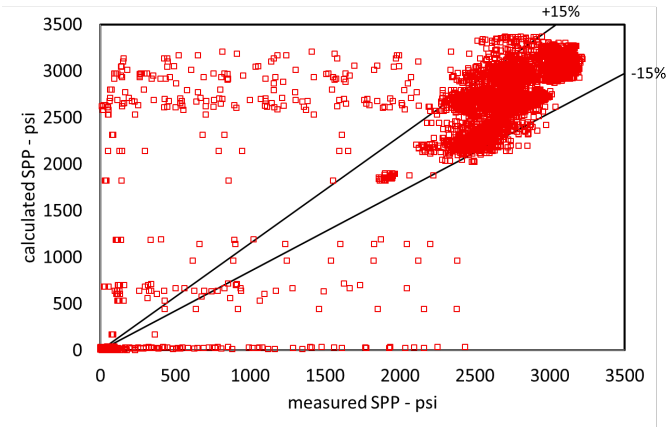


Fig. 2: *Physics-based model results of calculated vs. measured SPP values.*

modeled in agreement with drilling fluid behavior in the particular well geometry. Doing so, is key to be able to accurately predict the pressure losses such that well circulating pressures can be managed within the constraints of the so-called drilling margin.

Well contruction fluids (drilling fluid, completion fluids, cementing fluids etc.) are generally thixotropic non-Newtonian fluids that are shear-rate, temperature- and pressure-dependent. The rheological behavior of the drilling fluids is preferably modeled in accordance with the three-parameter Herschel and Bulkley fluid model ([HB26]), which is given by:

$$\tau = \tau_y + K\dot{\gamma}^m \tag{1}$$

where τ is shear stress, τ_y is yield stress, K is consistency index, $\dot{\gamma}$ is shear-rate and m is flow behavior index.

The SPP was predicted considering this rheological model and making use of an actual drilling dataset obtained for Well "A". This dataset pertains to a 4200 ft. drilling section and contains about 500K datapoints. In Fig. 1, the hole vs. bit depth curve with the SPP of Well A is presented.

3.1. Physics-based modeling

Flow in the circulation system during drilling can be summarized in three parts: flow in pipes (surface lines and inside the drillstring), annuli and the bit. Pump pressure (assuming no back-pressure applied on the annular side) is given as:

$$P_{pump} = \Delta P_{surface} + \Delta P_{drillstring} + \Delta P_{bit} + \Delta P_{annulus} \tag{2}$$

where P_{pump} is pump pressure, $\Delta P_{surface}$ is the pressure loss in the surface pipes, $\Delta P_{drillstring}$ is the pressure loss in the drillstring, ΔP_{bit} pressure loss at the bit and and $\Delta P_{annulus}$ is the pressure loss in the annulus. The standpipe pressure is measured at the downstream, high-pressure end of the pump, and can be approximated by the pump pressure while ignoring the minor frictional pressure loss contribution of the surface lines.

Physics-based modeling of the Herschel and Bulkley fluid flow at each individual geometry was accomplished with the equations presented in the literature ([BJMCYJ91], [ACM+09]). For the flow in annuli, the equations from ([EOM+15]) were used, which consider the effects of drillpipe eccentricity within the wellbore and rotation of the drillpipe on frictional pressure losses. The physics-based equations are derived from the first principles and were coded in Python. An iterative numerical scheme was programmed for the Herschel and Bulkley fluid flow in pipes and annuli. SciPy's scipy.optimize ([VGO+20]) was used to solve for the turbulent flow friction factor. Numba ([LPS15]) was used to accelerate the handling of computationally heavy functions.

At each time-step, a physics-based estimation was made for the entire dataset. Prediction performance, calculated vs. measured SPP of Well A is presented in Fig. 2.

The results show that the physics model underestimates the standpipe pressure, mainly because of the transient events occurring while turning the pumps on or off. An in-depth analysis suggests steady-state models estimate a zero pressure when the pumps are turned off. However, in reality, when the pumps are turned off, the pressure does not immediately drop to zero. It means that there is a delay between the flow rate and pressure, which is not accounted for in steady-state physics models.

3.2. Data-driven modeling

Deep learning neural networks perform very well in capturing the complex relationships of the data ([Hay94]). A PyTorch ([PGM+19]) implementation of a single and multi-layered neural network was developed to learn from the drilling time-series sensory data. The network was trained with flow rate, rotation rate, bit depth and hole depth to predict the standpipe pressure. Before training the network, the data was preprocessed and transformed using Scikit-Learn's preprocessing library ([PVG+11]). And, the training and test datasets are converted into NumPy ([Oli06]; [vCV11]) arrays.

While training the networks, Adam ([KB14]) was used as the optimizer for the model. At each epoch, a backward pass was made and the weights of the networks were updated. The Visdom ([vis]) library was used to visualize the loss function while the network was being trained. For most of the figures in this paper, the matplotlib library ([Hun07]) was used for visualization.

Several analyses were conducted to assess various neural network configurations to find feasible setups and good performance on drilling time-series data. First, the data was randomly shuffled and split using the PyTorch's random_split function to a 4:1 training to test ratio. Then, a single hidden layer neural network was trained. The performance results are presented in Fig. 3. Results show that even a single hidden layer neural network shows good performance, and that a 4:1 randomly shuffled learning provides a significant accuracy for this particular dataset. A quantitative analysis about the accuracy is presented in table 1.

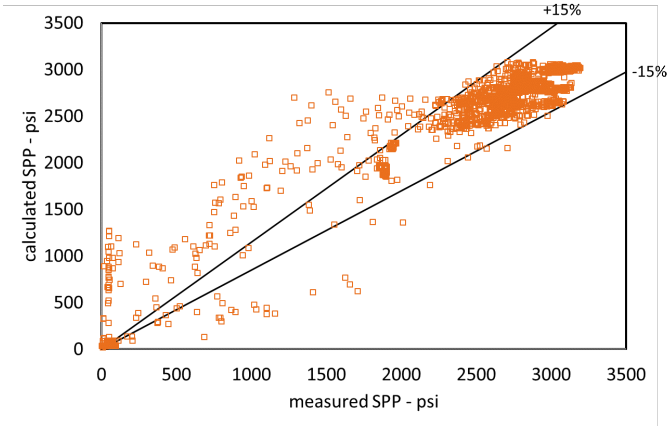


Fig. 3: *Data-driven model results of calculated vs. measured SPP values. Obtained by using neural networks, a single hidden layer, randomly sampling, and a 4-to-1 training-to-test ratio.*

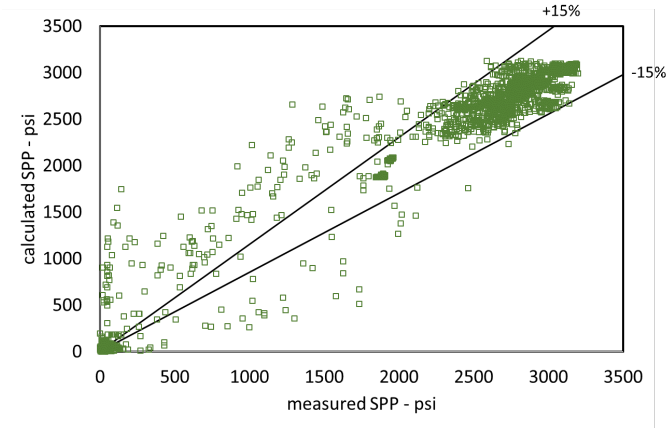


Fig. 4: *Data-driven model results of calculated vs. measured SPP values. Obtained by using neural networks, ten hidden layers, randomly sampling, and a 4-to-1 training-to-test ratio.*

For deep learning, the number of hidden layers were varied. It was found that approximately ten hidden layers appear to provide satisfactory results in terms of computational performance and accuracy for this particular drilling dataset. The results are shown in Fig. 4. In comparison with a single layer neural network, the accuracy was slightly better, and the time to train the neural network was shortened.

Additionally, a real-time system was assumed. Instead of randomly shuffling the data, the training data was divided into sequential intervals. The network was trained using the first two-thirds of the data to predict the standpipe pressure for the remaining one-third. The results are presented in Fig. 5. The network was able to identify the correlation of drilling parameters for the one-third progression of the drilling without the knowledge of the deeper sections. By only training from the initial two-thirds, the results still provided good performance for the latter one-third. The results show that even without randomly shuffling data and training the network, a good performance was obtained with this intervals-in-sequences approach.

Non-paramteric regression modeling using Gaussian Processes (GP) was also performed for this particular dataset. Scikit-Learn's

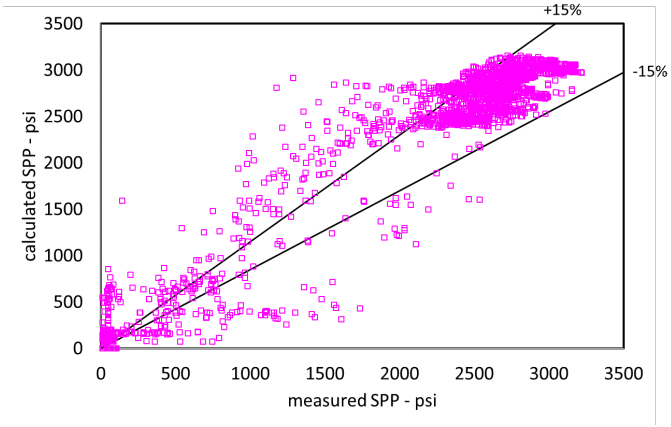


Fig. 5: *Data-driven model results of calculated vs. measured SPP values. Obtained by using neural networks, ten hidden layers, trained in sequential intervals. Learned first two-thirds of the dataset and predict the subsequent one-third.*

GP library was used with a Matérn kernel ([Ras]) as follows:

$$k_{Matérn} = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu r}}{\ell}\right)^{\nu} K_{\nu}\left(\frac{\sqrt{2\nu r}}{\ell}\right) \tag{3}$$

where ν and ℓ (length scale) are the hyperparameters of the kernel. . The parameter ν controls the smoothness of the learned function. In particular, the approximated function becomes smoother as the ν value gets larger. At $\nu = \infty$, the kernel becomes the Radial Basis Function (RBF) kernel.

For this particular drilling dataset, the priors and results of various kernels were analyzed. The Matérn kernel and a mixed kernel of RBF and WhiteKernel showed superior performance in comparison to others. Results from the Matérn kernel with $\nu = 0.5$ and $\ell = 1.0$ are presented.

Using the GPs, only a subsample of the entire drilling dataset was analyzed due to memory restrictions. The data was partitioned into so-called drilling stands, which consist of lenghts of three ~30 ft. drillpipe sections connected together. The reason for such stand-by-stand partitioning of the data was to get an abstract representation of the drilling process, and to localize the GPs training into drilling intervals.

In this example, the data from four historical stands were used in training the GPs to predict the SPP values for one stand into the future. The time-series results are shown in Fig. 6. Note that GPs also provided the cone of uncertainty with their prediction. The prediction performance is presented in Fig. 7. The results show that the predictions using GPs based on learning from the previous four stands show good agreement with the measurements.

3.3. Combination of the Physics-Based and Data-Driven Modeling

After obtaining the results from the physics-based and data-driven modeling, a rule-based stochastic decision-making algorithm was developed to combine these models. A hidden Markov model was constructed using the Pomegranate ([Sch17]) library.

Both the physics-based and data-driven models were combined with the process state of the operation and included in the hidden Markov model, as illustrated in Fig. 8. The process state was calculated by analyzing the multitude of sensor measurements (such as hookload, standpipe pressure, etc.) to analytically determine the

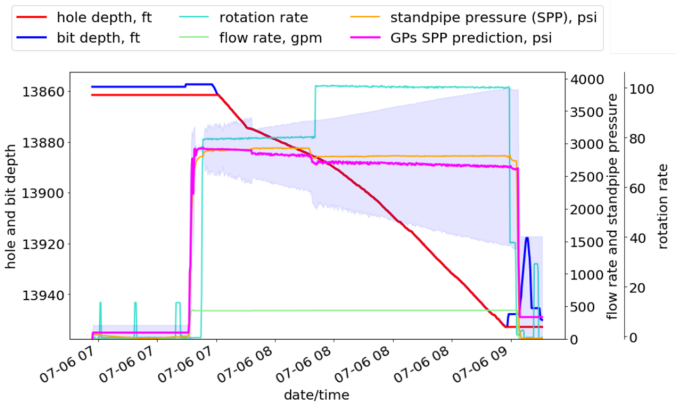


Fig. 6: *Data-driven model results presented in drilling time-series data of Well A. Obtained by using Gaussian Processes with a Matérn kernel with* $\nu = 0.5$ *and* $\ell = 1.0$.

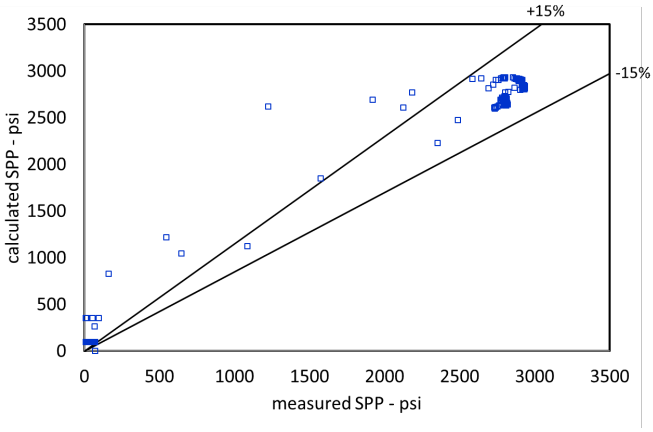


Fig. 7: *Data-driven model results of calculated vs. measured SPP values. Obtained by using Gaussian Processes with a Matérn kernel with* $\nu = 0.5$ *and* $\ell = 1.0$.

rig state ([HEC+19]). A simple pattern recognition technique, i.e. regular expressions ([Kle56]), which can be implemented through Python's re library or NumPy's ([Oli06]) numpy.where function, proved sufficient to calculate these rig states.

The hidden Markov model combined the information from the historical data, the process state, the physics-based and the data-driven model, following the flow chart shown in Fig. 8.

In this implementation of the hidden Markov model, the

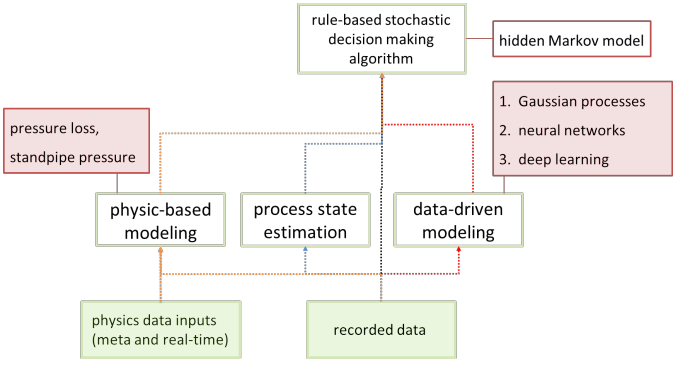


Fig. 8: *Combination of physics-based and data-driven modeling flow chart.*

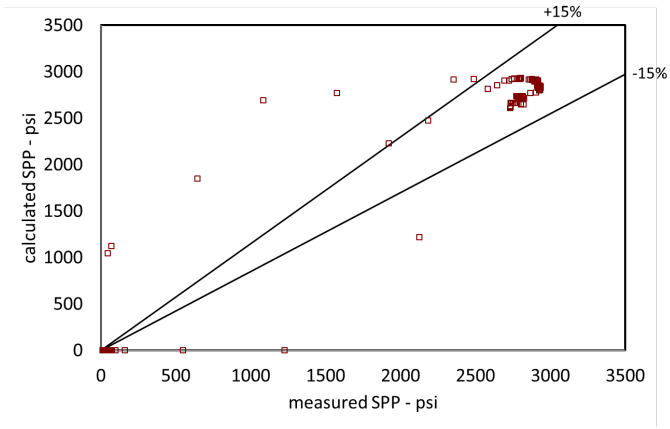


Fig. 9: *Proposed combined modeling results of calculated vs. measured SPP values. Physics-based model and GPs results are combined through a hidden Markov model.*

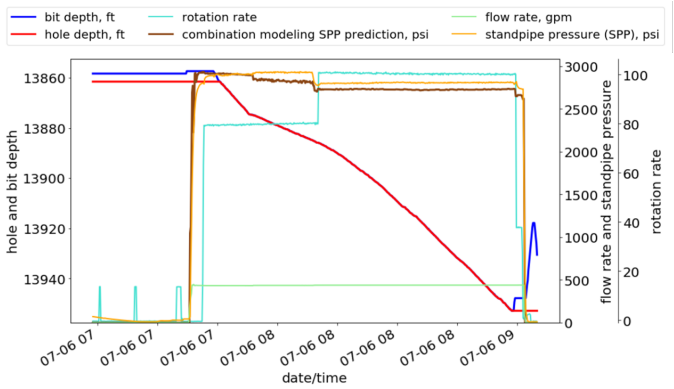


Fig. 10: *Combined modeling results presented in time-series drilling dataset.*

observable state is a string representation of the combined results from the physics-based and data-driven models and the rig's state. The hidden state is the combined ideal result given the circumstance.

Through the hidden Markov model some rules can be applied. A rule can for instance state that the likelihood of zero SPP can be significant for certain process state(s). For example, the SPP should be zero if drilling is temporarily halted and the drillstring is in slips with a zero flow rate. The Viterbi algorithm ([Vit67]) can then be used to calculate the path of the maximum likelihood.

As another example, we can identify inaccuracies of the physics-based model by understanding the operational state and historical SPP data. Doing so, yields an advantage to the data-driven model. The hidden Markov model will then attribute a higher weight and trust to the data-driven model. Comparatively, the results from the data-driven model can be unrealistic for various reasons (i.e. outliers, sensor errors, etc.), and the physics model can be assigned a higher weight.

By applying such rules, as an example, the combination modeling of physics and GPs model is achieved and the results are presented in Fig. 9.

The results showed good performance when the physics-based and data-driven models were combined. In Fig. 10, the results of the combination modeling in time-series is presented.

The hidden Markov model was able to provide better results

	RMSE	R^2	Median AE	Mean AE
Physics Model	619.4	0.699	93.7	256.6
NN [randomly sampled]	163.5	0.979	78.4	106.7
Deep Learning [randomly sampled]	158.0	0.980	60.5	92.1
Deep Learning [sequential interval]	213.9	0.963	148.7	172.2
Gaussian Processes	140.3	0.987	99.0	107.4
Combination model	109.4	0.992	74.8	71.4

TABLE 1: *Summary of the results from experimenting with various modeling approaches.*

through the application of relatively simple rules in comparison with using physics-based or data-driven modeling separately. The results show that the combined model outperformed all others. The summary of the results and their statistical significance are presented in the table 1.

4. Conclusions

A framework for combining physics-based and data-driven modeling is proposed through a rule-based stochastic decision-making algorithm. Physics-based modeling of standpipe pressure was performed using equations derived from first principles.. In addition, various data-driven modeling approaches were explored using a well dataset. Then, the two approaches were combined through the use of a hidden Markov model.

The combined model clearly outperforming all other models. Moreover, it managed to predict better results even while the pumps were off, a circumstance for which the data-driven model estimated unrealistic positive pressures.

Drilling critically relies on properly managing fluid circulating pressure in the wellbore for safety and efficiency. Through the proposed combination modeling, circulating pressure can be better predicted, which will lead to safer and more (cost-)efficient operations. Note that the proposed framework is not limited to the prediction of circulating pressure, and can be extended to other well construction domains.

Acknowledgment

The authors thank the Rig Automation and Performance Improvement in Drilling (RAPID) group at The University of Texas at Austin and its sponsors for their guidance and support.

Nomenclature

K : *consistency index*, $Pa\ s^m$
K_v : *modified Bessel function*
m : *flow behavior index*
P : *pressure*, Pa
k : *kernel function*

Greek Letters

τ : *shear stress*, Pa
$\dot{\gamma}$: *shear rate*, $1/s$
ν : *hyperparameter of Matérn kernel*
ℓ : *length scale of Matérn kernel*

Subscripts

y : *yield*

References

[ACK13] Dawn An, Joo Ho Choi, and Nam Ho Kim. Options for prognostics methods: A review of data-driven and physics-based prognostics. In *54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, page 1940, 2013. doi:10.2514/6.2013-1940.

[ACM+09] Bernt Aadnoy, Iain Cooper, Stefan Miska, Robert F Mitchell, and Michael L Payne. *Advanced drilling and well technology*. SPE Houston, TX, 2009.

[BJMCYJ91] Adam T Bourgoyne Jr, Keith K Millheim, Martin E Chenevert, and Farrile S Young Jr. Applied drilling engineering. 1991.

[DIX19] Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. Turbulence modeling in the age of data. *Annual Review of Fluid Mechanics*, 51:357–377, 2019. doi:10.1146/annurev-fluid-010518-040547.

[EOM+15] Oney Erge, Evren Mehmet Ozbayoglu, Stefan Miska, Mengjiao Yu, Nicholas Takach, Arild Saasen, Roland May, et al. The effects of drillstring-eccentricity,-rotation, and-buckling configurations on annular frictional pressure losses while circulating yield-power-law fluids. *SPE Drilling & Completion*, 30(03):257–271, 2015. URL: https://doi.org/10.2118%2F167950-pa, doi:10.2118/167950-pa.

[GSK19] Valeriy Gavrishchaka, Olga Senyukova, and Mark Koepke. Synergy of physics-based reasoning and machine learning in biomedical applications: towards unlimited deep learning with limited data. *Advances in Physics: X*, 4(1):1582361, 2019. doi:10.1080/23746149.2019.1582361.

[Hay94] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[HB26] Winslow H Herschel and Ronald Bulkley. Konsistenzmessungen von gummi-benzollösungen. *Kolloid-Zeitschrift*, 39(4):291–300, 1926. doi:https://doi.org/10.1007/BF01432034.

[HEC+19] Mohammad Hamzah, Oney Erge, Sylvain Chambon, et al. Automated drilling narratives: A scalable workflow to measure the effectiveness of drilling procedures. In *SPE/IADC International Drilling Conference and Exhibition*. Society of Petroleum Engineers, 2019. doi:https://doi.org/10.2118/194129-MS.

[Hun07] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. arXiv:1412.6980.

[Kle56] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3 – 42. Princeton University Press, Princeton, 1956. URL: https://princetonup.degruyter.com/view/book/9781400882618/10.1515/9781400882618-002.xml, doi:https://doi.org/10.1515/9781400882618-002.

[KSB17] Alina Kloss, Stefan Schaal, and Jeannette Bohg. Combining learned and analytical models for predicting action effects. *arXiv preprint arXiv:1710.04102*, 2017. arXiv:1710.04102.

[KWOI18] Tatsuya Kaneko, Ryota Wada, Masahiko Ozaki, and Tomoya Inoue. Combining physics-based and data-driven models for estimation of wob during ultra-deep ocean drilling. In *ASME 2018 37th International Conference on Ocean, Offshore and Arctic Engineering*. American Society of Mechanical Engineers Digital Collection, 2018. doi:https://doi.org/10.1115/OMAE2018-78229.

[KWRK17] Anuj Karpatne, William Watkins, Jordan Read, and Vipin Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling. *arXiv preprint arXiv:1710.11431*, 2017. arXiv:1710.11431.

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

[MM19] Daniel L Marino and Milos Manic. Combining physics-based domain knowledge and machine learning using variational gaussian processes with explicit linear prior. *arXiv preprint arXiv:1906.02160*, 2019. arXiv:1906.02160.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PGAAEMH19] Jorge Pulpeiro Gonzalez, King Ankobea-Ansah, Elena Escuder Milian, and Carrie M Hall. Modeling the gas exchange processes of a modern diesel engine with an integrated physics-based and data-driven approach. In *Dynamic Systems and Control Conference*, volume 59155, page V002T11A004. American Society of Mechanical Engineers, 2019. URL: https://doi.org/10.1115/DSCC2019-9226, arXiv:https://asmedigitalcollection.asme.org/DSCC/proceedings-pdf/DSCC2019/59155/V002T11A004/6455484/v002t11a004-dscc2019-9226.pdf, doi:10.1115/DSCC2019-9226.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, vier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[Ras] Carl Edward. Rasmussen. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass.

[RRS+18] Sk Rahman, Adil Rasheed, Omer San, et al. A hybrid analytics paradigm combining physics-based modeling and data-driven modeling to accelerate incompressible flow solvers. *Fluids*, 3(3):50, 2018. doi:10.3390/fluids3030050.

[Sch17] Jacob Schreiber. Pomegranate: fast and flexible probabilistic modeling in python. *The Journal of Machine Learning Research*, 18(1):5992–5997, 2017.

[vCV11] 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, 2011. doi:10.1109/MCSE.2011.37.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:https://doi.org/10.1038/s41592-019-0686-2.

[vis] visdom. https://github.com/facebookresearch/visdom. Accessed: 2020-05-13.

[Vit67] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967. doi:10.1109/TIT.1967.1054010.

Pydra - a flexible and lightweight dataflow engine for scientific analyses

Dorota Jarecka[‡*], Mathias Goncalves[¶‡], Christopher J. Markiewicz[¶], Oscar Esteban[¶], Nicole Lo[‡], Jakub Kaczmarzyk[§‡], Satrajit Ghosh[‡]

Abstract—This paper presents a new lightweight dataflow engine written in Python: *Pydra*. *Pydra* is developed as an open-source project in the neuroimaging community, but it is designed as a general-purpose dataflow engine to support any scientific domain. The paper describes the architecture of the software, as well as several useful features, that make *Pydra* a customizable and powerful dataflow engine. Two examples are presented to demonstrate the syntax and properties of the package.

Index Terms—dataflow engine, scientific workflows, reproducibility

Introduction

Scientific workflows often require sophisticated analyses that encompass a large collection of algorithms. The algorithms, that were originally not necessarily designed to work together, and were written by different authors. Some may be written in Python, while others might require calling external programs. It is a common practice to create semi-manual workflows that require the scientists to handle the files and interact with partial results from algorithms and external tools. This approach is conceptually simple and easy to implement, but the resulting workflow is often time consuming, error-prone and difficult to share with others. Consistency, reproducibility and scalability demand scientific workflows to be organized into fully automated pipelines. This was the motivation behind *Pydra* - a new dataflow engine written in Python, that is presented in this paper.

The *Pydra* package is a part of the second generation of the *Nipype* ecosystem ([GBM+11], [Dev]) --- an open-source framework that provides a uniform interface to existing neuroimaging software and facilitates interaction between different software components. The *Nipype* project was born in the neuroimaging community, and has been helping scientists build workflows for a decade, providing a uniform interface to such neuroimaging packages as FSL [WJP+09], ANTs [ATS09], AFNI [Cox96], FreeSurfer [DFS99] and SPM [FAK+07]. This flexibility has made it an ideal basis for popular preprocessing tools, such as fMRIPrep [OEG19] and C-PAC [C-P]. The second generation of *Nipype* ecosystem is meant to provide additional flexibility and is being developed with reproducibility, ease of use, and scalability in mind. *Pydra* itself is a standalone project and is designed as a general-purpose dataflow engine to support any scientific domain.

The goal of *Pydra* is to provide a lightweight dataflow engine for computational graph construction, manipulation, and distributed execution, as well as ensuring reproducibility of scientific pipelines. In *Pydra*, a dataflow is represented as a directed acyclic graph, where each node represents a Python function, execution of an external tool, or another reusable dataflow. The combination of several key features makes *Pydra* a customizable and powerful dataflow engine:

- **Composable dataflows:** Any node of a dataflow graph can be another dataflow, allowing for nested dataflows of arbitrary depths and encouraging creating reusable dataflows.
- **Flexible semantics for creating nested loops over input sets:** Any *Task* or dataflow can be run over input parameter sets and the outputs can be recombined (similar concept to Map-Reduce model [DG04], but *Pydra* extends this to graphs with nested dataflows).
- **A content-addressable global cache:** Hash values are computed for each graph and each Task. This supports reusing of previously computed and stored dataflows and *Tasks*.
- **Support for Python functions and external (shell) commands:** *Pydra* can decorate and use existing functions in Python libraries alongside external command line tools, allowing easy integration of existing code and software.
- **Native container execution support:** Any dataflow or *Task* can be executed in an associated container (via Docker or Singularity) enabling greater consistency for reproducibility.
- **Auditing and provenance tracking:** *Pydra* provides a simple JSON-LD -based message passing mechanism to capture the dataflow execution activties as a provenance graph. These messages track inputs and outputs of each task in a dataflow, and the resources consumed by the task.

Pydra is a pure Python 3.7+ package with a limited set of dependencies, which are themselves only dependent on the Python Standard library. It leverages *type annotation* and *AsyncIO* in its core operations. *Pydra* uses the *attr* package for extended annotation and validation of inputs and outputs of tasks, the *cloudpickle* package to pickle interactive task definitions, and the *pytest* testing framework. *Pydra* is intended to help scientific workflows which rely on significant file-based operations and

* *Corresponding author: djarecka@gmail.com*
‡ *Massachusetts Institute of Technology, Cambridge, MA, USA*
¶ *Stanford University, Stanford, CA, USA*
§ *Stony Brook University School of Medicine, Stony Brook, NY, USA*

which evaluate outcomes of complex dataflows over a hyper-space of parameters. It is important to note, that *Pydra* is not a framework for writing efficient scientific algorithms or for use in applications where caching and distributed execution are not necessary. Since *Pydra* relies on a filesystem cache at present, it is also not designed for dataflows that need to operate purely in memory.

The next section will describe the *Pydra* architecture --- main package classes and interactions between them. The *Key Features* section focuses on a set of features whose combination distinguishes *Pydra* from other dataflow engines. The paper concludes with a set of applied examples demonstrating the power and utility of *Pydra*, and short discussion on the future directions.

Architecture

Pydra architecture has three core components: *Task*, *Submitter* and *Worker*. *Tasks* form the basic building blocks of the dataflow, while *Submitter* orchestrates the dataflow execution model. Different types of *Workers* allow *Pydra* to execute the task on different compute architectures. Fig. 1 shows the Class hierarchy and interaction between them in the present *Pydra* architecture. It was designed this way to decouple *Tasks* and *Workers*. In order to describe *Pydra*'s most notable features in the next section, we briefly describe the role of each of these classes.

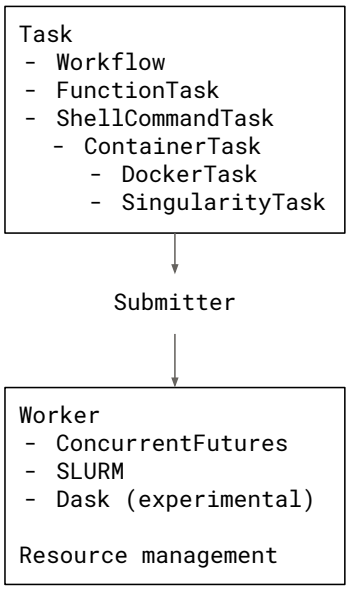


Fig. 1: *A schematic presentation of principal classes in Pydra.*

Dataflows Components: Task and Workflow

A *Task* is the basic runnable component of *Pydra* and is described by the class `TaskBase`. A *Task* has named inputs and outputs, thus allowing construction of dataflows. It can be hashed and executes in a specific working directory. Any *Pydra*'s *Task* can be used as a function in a script, thus allowing dual use in *Pydra*'s *Workflows* and in standalone scripts. There are several classes that inherit from `TaskBase` and each has a different application:

- `FunctionTask` is a *Task* that executes Python functions. Most Python functions declared in an existing library, package, or interactively in a terminal can be converted to a `FunctionTask` by using *Pydra*'s decorator - `mark.task`.

```
import numpy as np
from pydra import mark
fft = mark.annotate({'a': np.ndarray,
                     'return': float})(np.fft.fft)
fft_task = mark.task(fft)()
result = fft_task(a=np.random.rand(512))
```

fft_task is now a *PydraTask* and result will contain a *Pydra*'s `Result` object. In addition, the user can use Python's function annotation or another *Pydra* decorator—`mark.annotate` in order to specify the output. In the following example, we decorate an arbitrary Python function to create named outputs:

```
@mark.task
@mark.annotate(
    {"return": {"mean": float, "std": float}}
)
def mean_dev(my_data):
    import statistics as st
    return st.mean(my_data), st.stdev(my_data)

result = mean_dev(my_data=[...])()
```

When the *Task* is executed *result.output* will contain two attributes: *mean* and *std*. Named attributes facilitate passing different outputs to different downstream nodes in a dataflow.

- `ShellCommandTask` is a *Task* used to run shell commands and executables. It can be used with a simple command without any arguments, or with specific set of arguments and flags, e.g.:

```
ShellCommandTask(executable="pwd")

ShellCommandTask(executable="ls", args="my_dir")
```

The *Task* can accommodate more complex shell commands by allowing the user to customize inputs and outputs of the commands. One can generate an input specification to specify names of inputs, positions in the command, types of the inputs, and other metadata. As a specific example, FSL's BET command (Brain Extraction Tool) can be called on the command line as:

```
bet input_file output_file -m
```

Each of the command argument can be treated as a named input to the `ShellCommandTask`, and can be included in the input specification. As shown next, even an output is specified by constructing the *out_file* field form a template:

```
bet_input_spec = SpecInfo(
    name="Input",
    fields=[
    ( "in_file", File,
      { "help_string": "input file ...",
        "position": 1,
        "mandatory": True } ),
    ( "out_file", str,
      { "help_string": "name of output ...",
        "position": 2,
        "output_file_template":
                          "{in_file}_br" } ),
    ( "mask", bool,
      { "help_string": "create binary mask",
        "argstr": "-m", } ) ],
    bases=(ShellSpec,) )

ShellCommandTask(executable="bet",
                 input_spec=bet_input_spec)
```

Outputs can also be specified separately using a similar output specification.

- `ContainerTask` class is a child class of `ShellCommandTask` and serves as a parent class for `DockerTask` and `SingularityTask`. Both *Container Tasks* run shell commands or executables within containers with specific user defined environments using *Docker* [doc] and *Singularity* [sin] software respectively. This might be extremely useful for users and projects that require environment encapsulation and sharing. Using container technologies helps improve scientific workflows reproducibility, one of the key concept behind *Pydra*.
  These *Container Tasks* can be defined by using `DockerTask` and `SingularityTask` classes directly, or can be created automatically from `ShellCommandTask`, when an optional argument `container_info` is used when creating a *Shell Task*. The following two types of syntax are equivalent:

```
DockerTask(executable="pwd", image="busybox")

ShellCommandTask(executable="ls",
     container_info=("docker", "busybox"))
```

- `Workflow` - is a subclass of *Task* that provides support for creating *Pydra* dataflows. As a subclass, a *Workflow* acts like a *Task* and has inputs, outputs, is hashable, and is treated as a single unit. Unlike *Tasks*, workflows embed a directed acyclic graph. Each node of the graph contains a *Task* of any type, including another *Workflow*, and can be added to the *Workflow* simply by calling the `add` method. The connections between *Tasks* are defined by using so called *Lazy Inputs* or *Lazy Outputs*. These are special attributes that allow assignment of values when a *Workflow* is executed rather than at the point of assignment. The following example creates a *Workflow* from two *Pydra Tasks*.

```
# creating workflow with two input fields
wf = Workflow(input_spec=["x", "y"])
# adding a task and connecting task's input
# to the workflow input
wf.add(mult(name="mlt",
               x=wf.lzin.x, y=wf.lzin.y))
# adding anoter task and connecting
# task's input to the "mult" task's output
wf.add(add2(name="add", x=wf.mlt.lzout.out))
# setting worflow output
wf.set_output([("out", wf.add.lzout.out)])
```

State

All *Tasks*, including *Workflows*, can have an optional attribute representing an instance of the `State` class. This attribute controls the execution of a *Task* over different input parameter sets. This class is at the heart of *Pydra's* powerful *Map-Reduce* over arbitrary inputs of nested dataflows feature. The `State` class formalizes how users can specify arbitrary combinations. Its functionality is used to create and track different combinations of input parameters, and optionally allow limited or complete recombinations. In order to specify how the inputs should be split into parameter sets, and optionally combined after the *Task* execution, the user can set *splitter* and *combiner* attributes of the `State` class. These attributes can be set by calling `split` and `combine` methods in the *Task* class. Here we provide a simple *Map-Reduce* example:

```
task_with_state =
      add2(x=[1, 5]).split("x").combine("x")
```

In this example, the `State` class is responsible for creating a list of two separate inputs, *[{x: 1}, {x:5}]*, each run of the *Task* should get one element from the list. The results are grouped back when returning the result from the *Task*. While this example illustrates mapping and grouping of results over a single parameter, *Pydra* extends this to arbitrary combinations of input fields and downstream grouping over nested dataflows. Details of how splitters and combiners power *Pydra*'s scalable dataflows are described later.

Submitter

The `Submitter` class is responsible for unpacking *Workflows* and single *Tasks* with or without `State` into standalone stateless jobs, *runnables*, that are then executed by *Workers*. When the *runnable* is a *Workflow*, the *Submitter* is responsible for checking if the *Tasks* from the graph are ready to run, i.e. if all the inputs are available, including the inputs that are set to the *Lazy Outputs* from previous *Tasks*. Once a *Task* is ready to run, the *Submitter* sends it to a *Worker*. When the runnable has a *State*, then the *Submitter* unpacks the *State* and sends multiple jobs to the *Worker* for the same *Task*. In order to avoid memory consumption as a result of scaling of *Tasks*, each job is sent as a pointer to a pickle file, together with information about its state, so that proper input can be retrieved just before running the *Task*. *Submitter* uses *AsyncIO* to manage all job executions to work in parallel, allowing scaling of execution as *Worker* resources are made available.

Workers

Workers in *Pydra* are responsible for the actual execution of the *Tasks* and are initialized by the *Submitter*. *Pydra* supports three types of execution managers: *ConcurrentFutures*, *Slurm* and *Dask* (experimental). When `ConcurrentFuturesWorker` is created, `ProcessPoolExecutor` is used to create a "pool" for adding the runnables. `SlurmWorker` creates an'sbatch' submission script in order to execute the task, and `DaskWorker` make use of Dask's `Client` class and its `submit` method. All workers use *async functions* from *AsyncIO* in order to handle asynchronous processes. All *Workers* rely on a `load_and_run` function to execute each job from its pickled state.

Key Features

In this section, features of *Pydra* that exemplify its utility for scientific dataflows are presented. Individually, some of these features are present in the numerous workflow packages that exist, but *Pydra* is the only software that brings them together using a very lighweight codebase. The combination of the following features makes *Pydra* a powerful tool in scientific computation.

Nested and Hashed Workflows

Scientific dataflows typically involve significant refinement and extensions as science and instrumentation evolves. *Pydra* was designed to provide an easy way of creating scientific dataflows that range from simple linear pipelines to complex nested graphs. It enables reproducibility and reduces cost of dataflow maintenance through flexible reuse of already existing functions and *Workflows* in new applications. The `Workflow` class inherits

from `TaskBase` class and can be treated by users as any other *Task*, so can itself be added as a node in another *Workflow*. This provides an easy way of creating nested *Workflows* of arbitrary depth, and reuse already existing *Workflows*. This is schematically shown in Fig. 2.

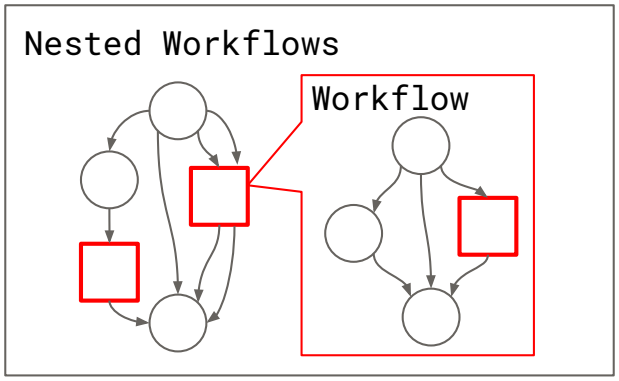


Fig. 2: *A nested Pydra Workflow, black circles represent single Tasks, and Workflows are represented by red rectangles.*

The *Pydra*'s *Submitter* supports this nested architecture and can dynamically extend the execution graph. Since a *Workflow* works like a *Task*—has inputs, outputs, and is hashable, once executed it does not need to recompute its operations if cached (*Pydra*'s caching is explained later in the section).

State and Nested Loops over Input

One of the main goals of creating *Pydra* was to support flexible evaluation of a *Task* or a *Workflow* over combinations of input parameters. This is the key feature that distinguishes it from most other dataflow engines. This is similar to the concept of the *Map-Reduce* [DG04], but extends it to work over arbitrary nested graphs. In complex dataflows, this would typically involve significant overhead for data management and use of multiple nested loops. In *Pydra*, this is controlled by setting specific `State` related attributes through *Task* methods. In order to set input splitting (or mapping), *Pydra* requires setting up a *splitter*. This is done using *Task*'s `split` method. The simplest example would be a *Task* that has one field x in the input, and therefore there is only one way of splitting its input. Assuming that the user provides a list as a value of x, *Pydra* splits the list, so each copy of the *Task* will get one element of the list. This can be represented as follow:

$$S = x : x = [x_1, x_2, ..., x_n] \longmapsto x = x_1, x = x_2, ..., x = x_n \ ,$$

where S represents the *splitter*, and x is the input field.

That is also represented in Fig. 3, where *x=[1, 2, 3]* as an example.

Scalar and outer splitters: Whenever a *Task* has more complicated inputs, i.e. multiple fields, there are two ways of creating the mapping, each one is used for different application. These *splitters* are called *scalar splitter* and *outer splitter* They use a special, but Python-based syntax as described next.

A *scalar splitter* performs element-wise mapping and requires that the lists of values for two or more fields to have the same length. The *scalar splitter* uses Python tuples and its operation is therefore represented by a parenthesis, `()`:

$$\begin{aligned} S = (x, y) \quad &: \quad x = [x_1, x_2, .., x_n], \ \ y = [y_1, y_2, .., y_n] \\ &\mapsto \quad (x, y) = (x_1, y_1), ..., (x, y) = (x_n, y_n), \end{aligned}$$

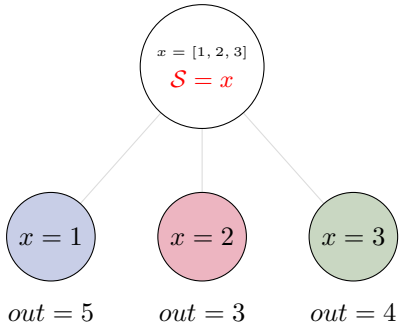


Fig. 3: *Diagram representing a Task with one input and a simple splitter. The white node represents an original Task with x=[1,2,3] as an input and S=x as a splitter. The coloured nodes represent stateless copies of the original Task after splitting the input, these are the runnables that are executed by Workers.*

where S represents the *splitter*, x and y are the input fields.

This is also represented as a diagram in Fig. 4

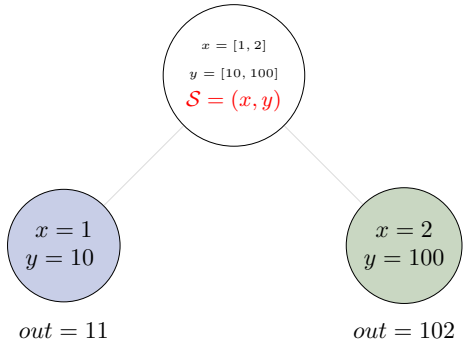


Fig. 4: *Diagram representing a Task with two input fields and a scalar splitter. The symbol convention is described in 3.*

The second option of mapping the input, when there are multiple fields, is provided by the *outer splitter*. The *outer splitter* creates all combination of the input values and does not require the lists to have the same lengths. The *outer splitter* uses Python's list syntax and is represented by square brackets, `[]`:

$$\begin{aligned} S = [x, y] \quad &: \quad x = [x_1, x_2, ..., x_n], \ \ y = [y_1, y_2, ..., y_m], \\ &\mapsto \quad (x, y) = (x_1, y_1), (x, y) = (x_1, y_2) ..., (x, y) = (x_n, y_m). \end{aligned}$$

The *outer splitter* for a node with two input fields is schematically represented in Fig. 5

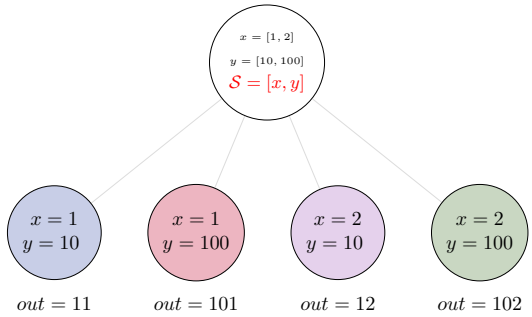


Fig. 5: *Diagram representing a Task with two input fields and an outer splitter. The symbol convention is described in 3.*

Different types of splitters can be combined over inputs such as *[inp1, (inp2, inp3)]*. In this example an *outer splitter* provides

all combinations of values of *inp1* with pairwise combinations of values of *inp2* and *inp3*. This can be extended to arbitrary complexity.

Combiners: In addition to the splitting the input, *Pydra* supports grouping or combining the output resulting from the splits. Taking as an example the simple *Task* represented in Fig. 3, in some application it can be useful to group all output values of the individual splits. In order to achieve this for a *Task*, a user can specify a *combiner*. This can be set by calling `combine` method. Note, the *combiner* only makes sense when a *splitter* is set first. When *combiner=x*, all values are combined together within one list, and each element of the list represents an output of the *Task* for the specific value of the input x. Splitting and combining for this example can be written as follows:

$$\begin{aligned} S = x \quad &: \quad x = [x_1, x_2, ..., x_n] \mapsto x = x_1, x = x_2, ..., x = x_n, \\ C = x \quad &: \quad out(x_1), ..., out(x_n) \mapsto out_{comb} = [out(x_1), ...out(x_n)], \end{aligned}$$

where S represents the *splitter*, C represents the *combiner*, x is the input field, $out(x_i)$ represents the output of the *Task* for x_i, and out_{comb} is the final output after applying the *combiner*.

In the situation where input has multiple fields and an *outer splitter* is used, there are various ways of combining the output. Taking as an example *Task* represented in Fig. 5, user might want to combine all the outputs for one specific value of :math:x_i and all the values of :math:y. In this situation, the combined output would be a two dimensional list, each inner list for each value of :math:x. This is written as follows:

$$\begin{aligned} C = y \quad &: \quad out(x_1, y1), out(x_1, y2), ...out(x_n, y_m) \\ &\longmapsto \quad [[out(x_1, y_1), ..., out(x_1, y_m)], \\ &\qquad ..., \\ &\qquad [out(x_n, y_1), ..., out(x_n, y_m)]]. \end{aligned}$$

And is represented in Fig. 6.

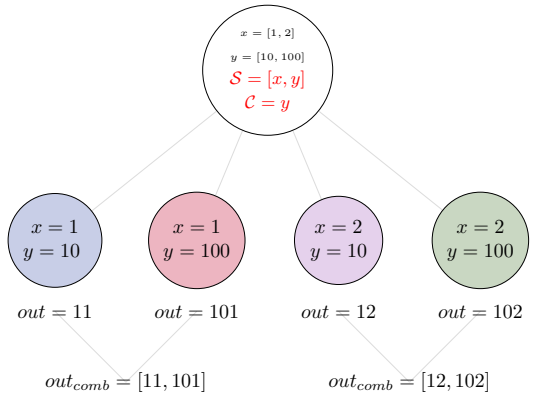


Fig. 6: *Diagram representing a Task with two input fields, an outer splitter and a combiner. The white node represents an original Task with x=[1,2], y=[10, 100] as an input, S=[x, y] as a splitter, and C=y as a combiner. The coloured nodes represent stateless copies of the original Task after splitting the input, these are the runnables that are executed by Workers. At the end outputs for all values of y are combined together within out_{comb}.*

However, for the diagram from 5, the user might want to combine all values of x for specific values of y. One may also need to combine all the values together. This can be achieved by providing a list of fields, *[x, y]* to the combiner. When a full combiner is set, i.e. all the fields from the splitter are also in the combiner, the output is a one dimensional list:

$$C = [x, y] : out(x_1, y1), ...out(x_n, y_m) \longmapsto [out(x_1, y_1), ..., out(x_n, y_m)].$$

And is represented in Fig. 7.

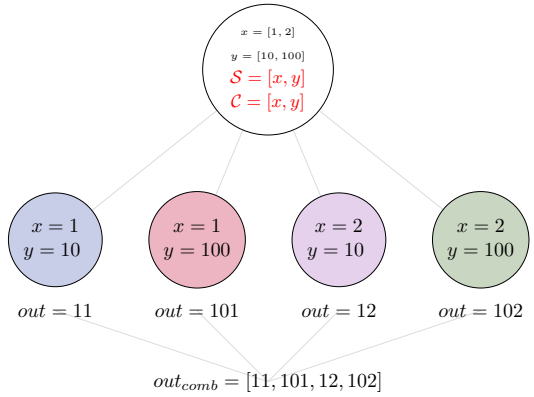


Fig. 7: *Diagram representing a Task with two input fields, an outer splitter and a full combiner. The Tasks are run in exactly the same way as previously, but at the end all of the output values are combined together. The symbol convention is described in 6.*

These are the basic examples of the *Pydra*'s *splitter-combiner* concept. It is important to note, that *Pydra* allows for mixing *splitters* and *combiners* on various levels of a dataflow. They can be set on a single *Task* or a *Workflow*. They can be passed from one *Task* to following *Tasks* within the *Workflow*. Examples of this more complex operation are presented in the next section.

Checksums and Global Cache

One of the key feature of *Pydra* is the support for a *Global Cache*. This allows multiple people in a laboratory, or even across laboratories to use each other's execution outputs on the same data without having to rerun the same computation. Each *Task* and *Workflow* has an attribute called *checksum*. In order to create the *checksum*, all of the input fields are collected and hash value is calculated. If *File* or *Directory* is used as an input, than the hash value of the content is used. For *Workflows*, the connections between the *Tasks* are also included in the final *checksum*, and hence the checksum of a *Workflow* changes if its underlying graph changes. The *checksum* is used to create output directory path during execution and can be reused in future executions of the same exact *Task* or *Workflow*. To reuse, a user can specify `cache_dir` and `cache_locations` when creating a *Task* or *Workflow*. The `cache_dir` is a read-write path, where you want your outputs to be saved, but `cache_location` can include a list of paths, which allow re-using existing caches. Before running any *Task* or *Workflow*, *Pydra* checks all the directories that are either in `cache_dir` or `cache_locations`, and if the specific checksum is found, then the results are reloaded without running the specific *Task*. It is important to emphasize that without a cache, every element of a nested *Workflow* would be re-executed. Using *Global Cache* can significantly reduce execution time when the same operations on the same data are repeated. This is also true for *Tasks* with *State*. If the number of input elements is expanded, the previously cached results can be reused without recomputation. For scientific workflows, where many tasks take significant computational resources, this can drastically speed up reruns.

Applications and Examples

In this section, we highlight *Pydra* through two examples. The first example is an intuitive scientific Python example to demonstrate the power of *Pydra*'s splitter and combiner. The second example extends this demonstration with a more practical machine learning model comparison workflow leveraging scikit-learn.

Example 1: Sine Function Approximation

This example illustrates the flexibility of the *Pydra*'s *splitters* and *combiners*, but the example is not meant to convince scientist to use *Pydra* to write algorithms like this. The exemplary workflow will calculate the approximated values of *Sine* function for various values of *x*. The *Workflow* uses the Taylor polynomial formula for *Sine* function:

$$\sum_{n=0}^{n_{max}} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$$

where n_{max} is a degree of approximation.

Since the idea is to make the execution as embarassingly parallel as possible, each of the term for each value of *x* should be calculated separately. This is done by function *term (x, n)*. In addition, *range_fun(n_max)* is used to return a list of integers from *0* to *n_max* and *summing(terms)* will sum all the terms for the specific value of *x* and *n_max*.

```
from pydra import Workflow, Submitter, mark
import math

@mark.task
def range_fun(n_max):
    return list(range(n_max+1))

@mark.task
def term(x, n):
    import math
    fract = math.factorial(2 * n + 1)
    polyn = x ** (2 * n + 1)
    return (-1)**n * polyn / fract

@mark.task
def summing(terms):
    return sum(terms)
```

The *Workflow* takes two inputs - a list of values of *x* and a list of values of *n_max*. In order to calculate various degrees of the approximation for each value of *x*, an *outer splitter* is used *[x, n_max]*. All approximations for a specific values of *x* is aggregated by using *n_max* as a combiner.

```
wf = Workflow(name="wf", input_spec=["x", "n_max"])
wf.split(["x", "n_max"]).combine("n_max")
wf.inputs.x = [0, 0.5 * math.pi, math.pi]
wf.inputs.n_max = [2, 4, 10]
```

All three *Function Tasks* are added to the *Workflow* and connected together using *lazy* connections. The second task, *term*, has to be additionally split over *n* to compute the different pieces of the Taylor approximation and the results of each term calculation are grouped together through the *combine* method.

```
wf.add(range_fun(name="range", n_max=wf.lzin.n_max))
wf.add(term(name="term", x=wf.lzin.x,
                         n=wf.range.lzout.out).
       split("n").combine("n"))
wf.add(summing(name="sum", terms=wf.term.lzout.out))
```

Finally, the *Workflow* output is set as the approximation using `set_output` method. Thus the *Workflow* reflects a parallelizable self contained function.

```
wf.set_output([("sin", wf.sum.lzout.out)])
res = wf(plugin="cf")
```

When executed using the concurrent futures library, the result is a two dimensional list of *Results*. For each value of *x* the *Workflow* computes a list of three approximations. As an example, for *x=pi/2* this returns the following list:

```
[...[Result(output=Output(sin=1.0045248555348174),
            runtime=None, errored=False),
     Result(output=Output(sin=1.0000035425842861),
            runtime=None, errored=False),
     Result(output=Output(sin=1.0000000000000002),
            runtime=None, errored=False)],
...]
```

Each *Result* contains three elements: *output* reflecting the actual computed output, *runtime* reflecting the information related to resources used during execution (when a resource audit flag is set), and *errored* a boolean flag which indicates whether the task errored or not. As expected, the values of the *Sine* function are getting closer to *1* with increasing degree of the approximation.

The described *Workflow* is schematically presented in Fig. 8.

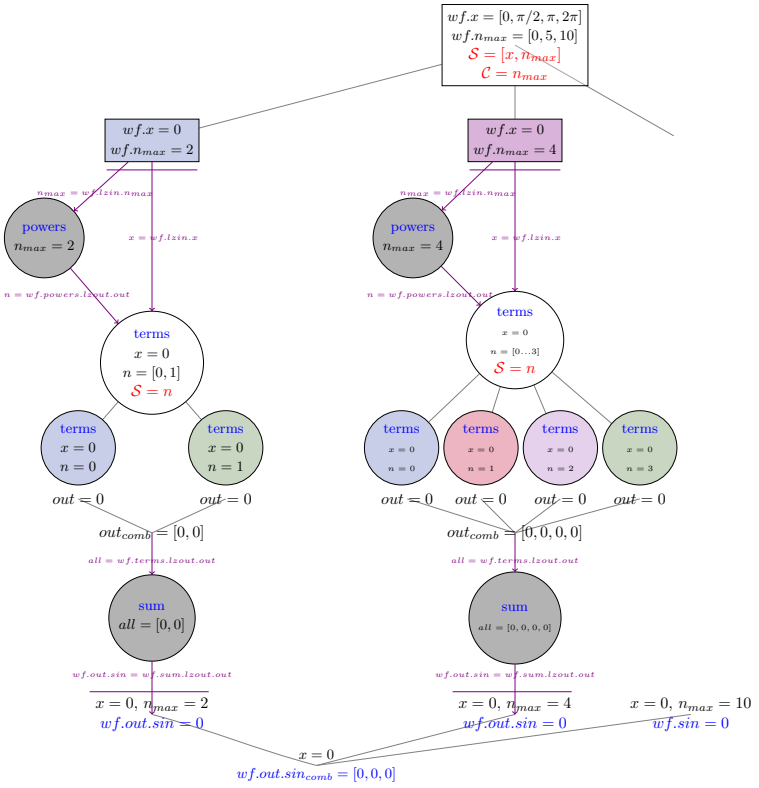


Fig. 8: *Diagram representing part of the Workflow for calculating* Sine *function approximations of various degrees for values of x. Circles represent single Tasks and rectangles represent Workflows. The white nodes represent Task or Workflow with a State. The coloured nodes represent stateless copies of the original Task after splitting the input. The gray nodes represent a Task that has no State.*

Example2: Machine Learning Model Comparison

The massive parameter search space of models and their parameters makes machine learning an ideal use case for *Pydra*. This section illustrates a general-purpose machine learning *Pydra*'s *Workflow* for model comparison using a boostrapped shuffle-split mechanism for choosing training and test pairs from a given dataset. The example leverages *Pydra*'s powerful splitters and

combiners to scale across a set of classifiers and metrics. It also uses *Pydra*'s caching to not redo model training and evaluation when new metrics are added, or when number of iterations is increased. The complete model comparison workflow is available as an installable package called *pydra-ml* [pyd], and includes SHAP-based feature importance evaluation in addition to model comparison.

The *Workflow* presented here comprises four *FunctionTasks*. For the sake of clarity, we will not redisplay the task code here. They can be found in the *tasks.py* file in *pydra-ml* [pyd]. The first function, *read_data*, reads csv data as a *pandas.DataFrame* and allows the user to extract specific columns as the input, *X*, to a learning model, a target column, *y*, and an optional *group* column. The second function, *gen_splits*, uses *GroupShuffleSplit* from *sklearn.model_selection* to generate a set of train-test splits given *n_splits* and *test_size*, with an option to define *group* and *random_state*. It returns *train_test_splits* and *split_indices*. The main function to train the classifier, *train_test_kernel*, takes as input a specific train-test split pair, a target variable, a parameter providing information about which classifier to use and whether to generate a null model by permuting the labels. The final function *calc_metric* returns the value from a scoring function given the actual target and predicted values from the classifier.

These tasks are combined together within a *Workflow* exploiting splitters and combiners. The *Workflow* itself has an *outer split* for *clf_info* and *permute*, allowing evaluation of null and non-null models for every classifier. The core model fitting and evaluation function *train_test_kernel* uses an internal splitter to iterate over all the bootstrapped iterations. Using *Pydra*, it is possible to split over *split_index*, that comes from *gensplit Task*, and run *train_test_kernel* for each of them without combining. This maintains `State` which can be used by the *calc_metric* function to evaluate different scoring methods on the classifier outputs and combine these results back together.

```
wf = pydra.Workflow(name="ml_wf",
          input_spec=list(inputs.keys()),
          **inputs,
          cache_dir=cache_dir,
          cache_locations=cache_locations)
# Workflow level splitting over combination
# of values
wf.split(["clf_info", "permute"])
wf.add(read_file(
        name="readcsv",
        filename=wf.lzin.filename,
        x_indices=wf.lzin.x_indices,
        target_vars=wf.lzin.target_vars))
wf.add(gen_splits(
        name="gensplit",
        n_splits=wf.lzin.n_splits,
        test_size=wf.lzin.test_size,
        X=wf.readcsv.lzout.X,
        Y=wf.readcsv.lzout.Y,
        groups=wf.readcsv.lzout.groups))
wf.add(train_test_kernel(
        name="fit_clf",
        X=wf.readcsv.lzout.X,
        y=wf.readcsv.lzout.Y,
        train_test_split=wf.gensplit.lzout.splits,
        split_index=wf.gensplit.lzout.split_indices,
        clf_info=wf.lzin.clf_info,
        permute=wf.lzin.permute))
# Task level splitting over bootstrapped
# train-test pairs
wf.fit_clf.split("split_index")
wf.add(calc_metric(
        name="metric",
        output=wf.fit_clf.lzout.output,
        metrics=wf.lzin.metrics))
# Downstream combination after calculating
# a set of metrics on each train-test pair
wf.metric.combine("fit_clf.split_index")
wf.set_output(
    [
     ("output", wf.metric.lzout.output),
     ("score", wf.metric.lzout.score),
     ("feature_names",
            wf.readcsv.lzout.feature_names),
    ]
)
```

The workflow is executed by providing an input dictionary exemplary input dictionary and the *Workflow*'s submission can look as follow:

```
clfs = [
 ('sklearn.ensemble', 'ExtraTreesClassifier',
  dict(n_estimators=100)),
 ('sklearn.neural_network', 'MLPClassifier',
  dict(alpha=1, max_iter=1000)),
 ('sklearn.neighbors', 'KNeighborsClassifier', dict(),
 [{'n_neighbors': [3, 7, 15],
   'weights': ['uniform','distance']}]),
 ('sklearn.ensemble', 'AdaBoostClassifier', dict())]

inputs = {"filename": 'iris.csv',
          "x_indices": range(4), "target_vars": ("label"),
          "n_splits": 3, "test_size": 0.2,
          "metrics": ["roc_auc_score"],
          "permute": [True, False], "clf_info": clfs}
n_procs = 8 # for parallel processing
cache_dir = os.path.join(os.getcwd(), 'cache')
wf_cache_dir = os.path.join(os.getcwd(), 'cache-wf')

# Execute the workflow in parallel using multiple processes
with pydra.Submitter(plugin="cf", n_procs=n_procs) as sub:
    sub(runnable=wf)

result = wf.result(return_inputs=True)
```

The result from the *Workflow* is a set of scores for permuted and non-permuted models. This is a list, each element of the list is for one value of *clf_info* and *permute*, both fields were set as input fields to the *Workflow*. All *Result* objects have an *output.score* field that is also a list. Each element of the *score* corresponds to a different value of *split_index*, that was set both as a *splitter* and *combiner* to the *fit_cls Task*. This gives an option to easily compare various models and sets of parameters.

```
[({'ml_wf.clf_info':
      ('sklearn.ensemble','ExtraTreesClassifier',
       {'n_estimators': 100}),
  'ml_wf.permute': True},
 Result(output=Output(score=[0.2622, 0.1733, 0.2975]),
        runtime=None, errored=False)),
({'ml_wf.clf_info':
       ('sklearn.ensemble', 'ExtraTreesClassifier',
        {'n_estimators': 100}),
  'ml_wf.permute': False},
 Result(output=Output(score=[1.0, 0.9333, 0.9333]),
        runtime=None, errored=False)),

...

({'ml_wf.clf_info':
      ('sklearn.ensemble', 'AdaBoostClassifier', {}),
  'ml_wf.permute': False},
 Result(output=Output(score=[0.9658, 0.9333, 0.8992]),
        runtime=None, errored=False))]
```

Usually, there is no easy way in *scikit-learn* to compare models in parallel across a variety of classifiers without using loops. It is possible to do all this natively in scikit-learn and joblib, but would

require much more code to do the maintenance of the dataflow and aggregation.

Summary and Future Directions

Pydra is a new lightweight dataflow engine written in Python. The combination of several key features - including flexible option for splitting and combining input fields, and *Global Cache* - makes *Pydra* a customizable and powerful dataflow engine. The *Pydra*'s developers are mostly from the Neuroimaging community, which provides a plethora of use-cases for complex dataflows, but the package is designed as a general-purpose dataflow engine to support any scientific domain. As the next step, the developer team would like to invite more scientist to use *Pydra* in order to test the package for diverse applications. In the near future, the developer team is also planning to work on:

- improvement of *Worker* classes to coordinate resource management
- improved interaction with *Dask* and other resource managers (e.g., SLURM) in HPC and Cloud environments.
- updates to the *Nipype* software to use *Pydra* as its engine
- improve the documentation and tutorials

We welcome scientists and developers to join the project. The project repository is available on GitHub under *Nipype* organization: https://github.com/nipype/pydra. In addition, there is also a repository that contains Jupyter Notebooks with *Pydra* tutorial: https://github.com/nipype/pydra-tutorial. The tutorial can be run locally or using the Binder service.

Acknowledgements

This was supported by NIH grants P41EB019936, R01EB020740. We thank the neuroimaging community for feedback during development, and Anna Jaruga for her feedback on the paper.

REFERENCES

[ATS09] Brian B Avants, Nick Tustison, and Gang Song. Advanced normalization tools (ants). *Insight j*, 2(365):1–35, 2009.

[C-P] C-PAC. http://fcp-indi.github.io/.

[Cox96] Robert W. Cox. Afni: Software for analysis and visualization of functional magnetic resonance neuroimages. *Computers and Biomedical Research*, 29(3):162 – 173, 1996. URL: http://www.sciencedirect.com/science/article/pii/S0010480996900142, doi:https://doi.org/10.1006/cbmr.1996.0014.

[Dev] Nipype Developers.

[DFS99] Anders M. Dale, Bruce Fischl, and Martin I. Sereno. Cortical surface-based analysis: I. segmentation and surface reconstruction. *NeuroImage*, 9(2):179 – 194, 1999. URL: http://www.sciencedirect.com/science/article/pii/S1053811998903950, doi:https://doi.org/10.1006/nimg.1998.0395.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[doc] Docker. https://www.docker.com/.

[FAK+07] K.J. Friston, J. Ashburner, S.J. Kiebel, T.E. Nichols, and W.D. Penny, editors. *Statistical Parametric Mapping: The Analysis of Functional Brain Images*. Academic Press, 2007. URL: http://store.elsevier.com/product.jsp?isbn=9780123725608.

[GBM+11] Krzysztof Gorgolewski, Christopher Burns, Cindee Madison, Dav Clark, Yaroslav Halchenko, Michael Waskom, and Satrajit Ghosh. Nipype: A flexible, lightweight and extensible neuroimaging data processing framework in python. *Frontiers in Neuroinformatics*, 5:13, 2011. URL: https://www.frontiersin.org/article/10.3389/fninf.2011.00013, doi:10.3389/fninf.2011.00013.

[OEG19] Ross W. Blair Craig A. Moodie A. Ilkay Isik Asier Erramuzpe James D. Kent Mathias Goncalves Elizabeth DuPre Madeleine Snyder Hiroyuki Oya Satrajit S. Ghosh Jessey Wright Joke Durnez Russell A. Poldrack Oscar Esteban, Christopher J. Markiewicz and Krzysztof J. Gorgolewski. fmriprep: a robust preprocessing pipeline for functional mri. *Nature Methods*, 16:111 – 116, 2019. doi:doi:10.1038/s41592-018-0235-4.

[pyd] pydra-ml. https://github.com/nipype/pydra-ml.

[sin] Singularity. https://sylabs.io/docs/.

[WJP+09] Mark W. Woolrich, Saad Jbabdi, Brian Patenaude, Michael Chappell, Salima Makni, Timothy Behrens, Christian Beckmann, Mark Jenkinson, and Stephen M. Smith. Bayesian analysis of neuroimaging data in fsl. *NeuroImage*, 45(1, Supplement 1):S173 – S186, 2009. Mathematics in Brain Imaging. URL: http://www.sciencedirect.com/science/article/pii/S1053811908012044, doi:https://doi.org/10.1016/j.neuroimage.2008.10.055.

Leading magnetic fusion energy science into the big-and-fast data lane

Ralph Kube[¶*], R Michael Churchill[¶], Jong Youl Choi[§], Ruonan Wang[§], Scott Klasky[§], CS Chang[¶], Minjun J. Choi[‡], Jinseop Park[‡]

https://youtu.be/rih7Hp9nPvM

Abstract—We present `Delta`, a Python framework that connects magnetic fusion experiments to high-performance computing (HPC) facilities in order leverage advanced data analysis for near real-time decisions. Using the ADIOS I/O framework, `Delta` streams measurement data with over 300 MByte/sec from a remote experimental site in Korea to Cori, a Cray XC-40 supercomputer at the National Energy Energy Research Scientific Computing Centre in California. There `Delta` dispatches cython data analysis kernels using an mpi4py PoolExecutor in order to perform a spectral data analysis workflow. Internally `Delta` uses queues and worker threads for data communication. With this approach we perform a common spectral analysis suite on imaging measurements more than 100 times faster than with a single-core implementation.

Index Terms—streaming analysis, mpi4py, queue, adios, HPC

Magnetic Fusion Energy research and its data analysis needs

Research on magnetic fusion energy combines physics, engineering, and even economics to deploy a virtually unlimited, clean, and competitively priced energy source to the grid. Python is well established in the fusion community through projects like plasmapy [PPY] or OMFIT [Men15]. We introduce another Python library for fusion energy reserch, `Delta` - the aDaptive nEar-reaL Time Analysis framework - and show how it can be used to stream data from an experiment to a remote high performance computing (HPC) resource [git].

There, Delta executes a routine spectral analysis workflow in near real-time. By making data analysis results available in near real-time, Delta allows scientists to make more informed decisions on follow-up experiments and could accelerate scientific discovery. To illustrate the use-case for Delta in fusion energy research, we start with a primer of fusion energy, introduce tokamak devices that are used to perform fusion experiments, describe a diagnostic that is installed in many tokamaks. With this at hand, we describe how near real-time data analysis can be used to accelerate experimental fusion energy workflows.

If one could harvest the energy from controlled nuclear fusion reactions you would have a potentially unlimited, environmentally friendly energy source. Fusion reactions release energy when two light nuclei merge into a heavier one. As part of the reaction, a fraction of the reactants nuclear binding energy is converted into kinetic energy of the products. Fission reactions on the other hand, which power todays nuclear power plants, release binding energy when a heavy nucleus decays into lighter products. Typical energies involved in nuclear reactions are measured in MeV, multiple orders of magnitude larger than the characteristic eV energy scale for chemical reactions. Thus, the energy yield for a nuclear reaction is much larger than for chemical reaction, which occur when fossil fuels are burnt. Fuel for fusion reactions are readily extracted from sea water, which is available in virtually inexhaustible quantities. Since the energy yield of a fusion reaction is so large, only little fusion plasma needs to be confined to power a fusion reactor. To produce 1 GW of fusion power, enough to power about 700,000 homes, just 2 kg of fusion plasma would need to be burned per day [Ent18]. Thus, a catastrophic event such as total loss of plasma confinement can cause no more than local damage to the plasma vessel.

To fuse positively charged atoms into one heavier requires enormous energy. For the most feasible fusion reactions, Deuterium-Tritium, temperatures upwards of 100 million degrees are required. Such a requirement unfortunately excludes any material container to confine a fusion fuel. The most promising approach is to confine the fusion fuel in the state of a plasma - a hot gas where the atoms are stripped of their electrons. Such a plasma can be confined in a strong magnetic field, shaped like a donut. Confined like this, there is no possibility for an uncontrolled chain reaction. If a significant amount of plasma would leak out of the vessel, the accompanying temperature drop would stop any fusion reactions. At the same time there are only a few grams of plasma confined and it does not have enough stored energy to cause damage other than to the structure of the confinement vessel.

The best performing plasma confinement devices, tokamaks, have a toroidal shape, similar to a donut. Tokamaks (a transliteration of the Russian acronym for toroidal chamber with magnetic coils), such as KSTAR [KSTAR] have a major radius R=1-1.5m and a minor radius a=0.2-0.7m. In experiments at these facilities, researchers configure parameters such as the plasma density or the shaping and strength of the magnetic field and study the behaviour of the plasma in this setup. During a typical experimental workflow, about 20-30 plasma discharges, so-called `shots` are performed on a given day where each shot lasts for

* *Corresponding author: rkube@pppl.gov*
¶ *Princeton Plasma Physics Laboratory*
§ *Oak Ridge National Laboratory*
‡ *National Fusion Research Institute, Daejeon 34133, Republic of Korea*

Task	Time-scale
real-time control	millisecond
live/inter-shot analysis	seconds, minutes
scientific discovery	hours, days, weeks

TABLE 1: *Time-scales on which analysis results of fusion data is required for different tasks.*

a couple of seconds up to minutes. Numerous measurements of the plasma and the mechanical components of the tokamak are performed during each discharge. After a cool-down phase of a few minutes (tokamaks contain cryogenic components) the device is ready for the next shot.

A common diagnostic in magnetic fusion experiments is a so-called Electron Cyclotron Emission (ECE) diagnostic [Cos74]. They measure emission intensity by free electrons in the plasma, which allows one to infer their temperature as a function of radius. Physical models of the plasma describe it partially through the temperature. This measurement allows one to interpret the experiment in terms of such models. Modern ECE systems, such as the one installed in the KSTAR tokamak [Yun14] have hundreds of spatial channels and sample data on a microsecond time-scale, producing data streams upwards of 500 MB/sec.

Analyzing large datasets, as produced by ECE diagnostics in between shots and generating actionable information in time for the next shot is a challenging task. The `Delta` framework aims to facilitate the analysis of such large datasets in near real-time." This use-case falls in between two other common data analysis workflows in fusion energy research, listed in Tab. 1. Real-time control systems for plasma control require data on a millisecond time scale. This time scale is a hard constraint and limits the amount of data the algorithms can ingest. Post-shot batch analysis of measurements on the other hand serves scientific discovery, such as extraction about the plasma from ECE data. The data and the analysis methods are selected on a per-case basis and are often performed manually hours, days, weeks, months, or years after an experiment has concluded. A goal of Delta is to facilitate scientific discovery at time-scale faster than the experimental cadence. Providing timely analysis results of plasma measurements to experimentalists aids them in making informed decisions about the next plasma shot. As an example of the workflows that we wish to facilitate with `Delta` we refer to a series of experiments performed at the TAE facility [Bal17]. There, the so-called `optometrist` algorithm was used as a stochastic optimizer in conjunction with expert judgement of domain scientists to assess the performance of a just concluded plasma shot and optimize the machine parameters in order to increase the performance of the following shot. By making advanced data analysis results available in near real-time to domain scientists, `Delta` will allow to improve workflows at experimental fusion facilities.

Designing the Delta framework

We are designing the `Delta` framework in a bottom-up approach, tailoring it to facilitate a specific spectral analysis workflow that uses measurements from an ECEI diagnostic. While plasma diagnostics operated at fusion experiments produce a heterogeneous set of data streams, the ECEI spectral analysis workflow is representative for a large set of workflows used to analyze different measurements. HPC environments also differ for example in their local area network topologies, the speed of network links between data-transfer nodes to compute node, compute node interconnects, and their network security policies. Furthermore granted allocations of compute time for research projects make it impractical to start with a top-down approach that generalizes well to arbitrary HPC platforms (though we endeavor to build the framework with flexibility and extensibility in mind). In the remainder of this section we describe the data analysis workflow for ECEI data, the targeted network and deployment architecture and give an overview of how `Delta` connects them together.

Electron Cyclotron Emission Imaging

The Electron Cyclotron Emission Imaging diagnostic installed in KSTAR measures the electron temperature T_e on a 0.15m by 0.5m grid, resolved using 8 horizontal and 24 vertical channels [Yun10], [Yun14]. Each individual channel produces an intensity time series $I_{h,v}(t_i)$ where h and v index the horizontal and vertical channel number and $t_i = i\Delta_t$ denotes the time where the intensity is sampled with $\Delta_t \approx 1\mu s$ being the sampling time. Digitized with a 16-bit digitizer, this diagnostic produces a data stream of 1836 MByte/sec. The spatial view of this diagnostic covers a significant area of the plasma cross-section which allows it to directly visualize the large-scale structures of the plasma. Besides analyzing the normalized intensity, several quantities calculated off the Fourier transformed intensity $X(\omega)$, here ω denotes the angular frequency, are used to study the plasma dynamics. The cross-power S, the coherence C, the cross-phase P and the cross-correlation R are defined respectively for channel pair combinations of Fourier transformed intensity signals X and Y as

$$S_{xy}(\omega) = E[X(\omega)Y^{\dagger}(\omega)], \tag{1}$$

$$C_{xy}(\omega) = |S_{xy}(\omega)|/\sqrt{S_{xx}(\omega)}/\sqrt{S_{yy}(\omega)}, \tag{2}$$

$$P_{xy}(\omega) = arctan(Im(S_{xy}(\omega))/Re(S_{xy}(\omega)), \tag{3}$$

and

$$R_{xy}(t) = IFFT(S_{xy}(\omega)). \tag{4}$$

Here E denotes an ensemble average, † denotes complex conjugation, Re and Im denote the real and imaginary part of a complex number and $IFFT$ denotes the inverse Fourier transform. In practice we use a short-time Fourier transformation (STFT) which averages the Fourier coefficients obtained from FFTs calculated on slightly shifted time windows. Spectral quantities calculated off local T_e fluctuations, such as the cross coherence or the cross phases, are used to identify macro-scale structures, so called magnetic islands, as well as micro-scale instabilities in the plasma [Cho17]. Understanding the physics resulting in magnetic islands is important for plasma confinement, and avoiding sudden loss of plasma control, known as a disruption.

Targeted HPC architecture

We implement `Delta` for streaming data from KSTAR to the National Energy Research Scientific Computing Centre (NERSC). NERSC operates Cori [cori], a Cray XC-40 supercomputer that is comprised of 2,388 Intel Xeon "Haswell" processor nodes, 9,688 Intel Xeon Phi "Knight's Landing" (KNL) nodes and ranks 16 on the Top500 list [top500]. Figure 1 illustrates the targeted network topology. Data transfers from KSTAR and NERSC originate and end at their respective Data Transfer Node (DTN). DTNs

Where	CPU	RAM	NIC
KSTAR DTN	Xeon E5-2640 v4	128GB	100 Gbit
NERSC DTN	Xeon E5-2680 v2	128GB	2 * 100 Gbit 2 * 10 Gbit
Cori compute	Xeon E5-2698 v3 32 threads	128GB	Cray Aries

TABLE 2: *Hardware and network interconnections of the data transfer nodes (DTNs) and compute nodes*

are servers deployed and configured specifically for transferring data over networks [dtn]. As such, they feature large-bandwidth network interfaces, both for internal and external connections. Table 2 lists the hardware of the DTNs and Cori's compute nodes. 100Gbit/s links connect both DTNs via the internet. At NERSC, the DTN is connected to Cori with dual 10 Gbit/s NICs. Coris individual compute nodes are connected with a Cray Aries interconnect, peaking at > 45 TB/s.

Connecting science experiments to HPC resources

`Delta` uses three separate software components to connect KSTAR to Cori. A **generator** running on the KSTAR DTN, a **middleman** running on the NERSC DTN, and a **processor** running on Cori. To avoid performance hits when touching the filesystem, `Delta` streams data between its components. By using only in-memory data transfers this avoids performance hits when touching the file system. Performance variability of file-based I/O on HPC systems is however a well studied problem [Xie12] . Furthermore, the workflow at hand and anticipated workflows require frequent transfer of relatively small data packets. By relying on streaming data transfers, `Delta` aims to avoid jitter due to I/O bottlenecks.

The generator stages measurement data and sends it to the middleman. The middleman forwards the data to the processor. The processor receives the data, executes the appropriate analysis kernels and stores the analysis results. `Delta` uses ADIOS [adios] to facilitate high bandwidth streaming on the paths marked with orange arrows in 1. ADIOS is a unified input/output system that transports and transforms groups of self-describing data variables across different media with performance as a main goal. It is commonly used in the Department of Energy for high performance parallel I/O. Its transport interface is step-based, which resembles the generation of scientific data. ADIOS implements multiple transport mechanisms as engines which take advantage of underlying network communication mechanisms to provide optimal performance. For the topology at hand, `Delta` configures ADIOS to use the DataMan engine for both, trans-oceanic data and intra-datacenter transfer. Switching the engine used by ADIOS is trivial and requires only the change of a single line in a configuration file.

Implementaion details

After providing an overview of the `Delta` framework and introducing its components in the previous section we continue by describing their implementation in the this section and benchmark their performance in the following section.

Components of the `Delta` framework

We continue by discussing the architecture of the `generator` and the `processor` shown in Fig. 2. Here we omit the `middleman` as it only relays the data stream.

The `generator` is a single-threaded application that sources data using a `loader` instance which handles all diagnostic specific data transformations and stages it for streaming. For the ECEI diagnostic the tranformations includes for example calculating a channel-dependent normalization and the aggregation of data into time chunks, N_{ch} blocks of s_{ch} consecutive samples. A `writer` instance handles all streaming I/O using ADIOS. Pseudo-code for the generator looks like this:

```
loader = loader_ecei(cfg["ECEI"])
writer = writer_gen(cfg["transport_tx"])
writer.Open()

batch_gen = loader.batch_generator()
for batch in batch_gen:
    writer.BeginStep()
    writer.put(batch)
    writer.EndStep()
```

Here, cfg is a framework-wide json configuration file. Diagnostic-specific parameters are stored in the `ECEI` section. ADIOS parameters for the writer are stored in the `transport_tx` section. Since all data transformations are applied inside the loader class the generator appears to be agnostic about what kind of diagnostic data it gets. However, the number of data batches is specific to the data at hand. Furthermore, the pseudo-code example above demonstrates the step-centered design of the ADIOS library. It encapsulates each time chunk in a single time step.

The middleman runs on the NERSC DTN and relays data streams from the generator to the processor. Using the classes available in `Delta` the pseudo-code looks similar to the `generator` with a `reader` instance replacing the `loader`..

The `processor` runs on Cori. It reads time chunks from an ADIOS stream, publishes them in a queue and submits analysis tasks to a pool of worker threads. As illustrated in Fig. 2 a `reader` object receives time chunks data and passes them to a `task_list` instance, which submits all data analysis routines on the `executors`. Pseudo-code for the processor looks like this

```
def consume(Q, task_list):
  while True:
    try:
      msg = Q.get(timeout=timeout_in_secs)
    except queue.Empty:
      break
    task_list.submit(msg)
  Q.task_done()


def main():
  executor_fft = MPIPoolExecutor(max_workers=NF)
  executor_anl = MPIPoolExecutor(max_workers=NA)
  a2_reader = reader(cfg["transport_rx"])
  a2_reader.Open()
  task_list = task_list_spectral(executor_anl,
                                 executor_fft,
                                 cfg)

  dq = Queue.Queue()
```

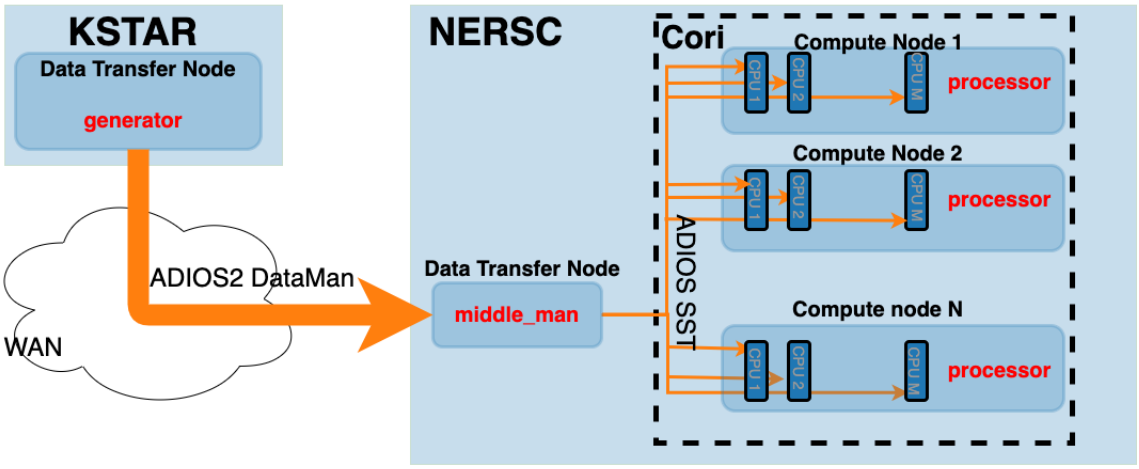


Fig. 1: *The network topology for which the* `Delta` *framework is designed. Data is streamed in the direction indicated by the orange arrow. At KSTAR, measurement data is staged from its DTN to the NERSC DTN. Internally at NERSC, the data stream is forwarded to compute nodes at Cori and analyzed. Orange arrows mark sections of the network where ADIOS facilitates high-performance streaming. Black arrows denote standard TCP/IP connections.*

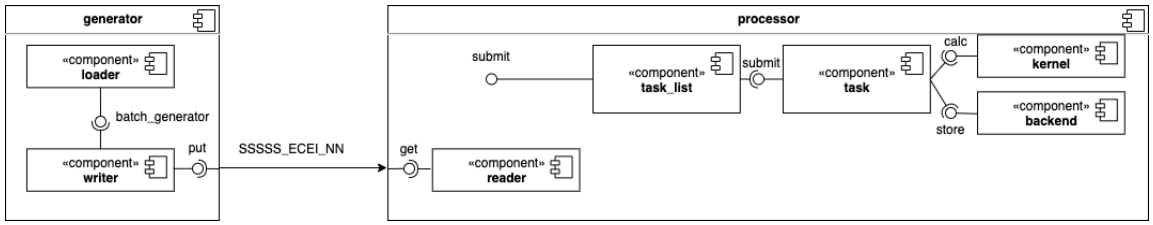


Fig. 2: *Schematic of the* `Delta` *framework. The* ***generator*** *runs at the data staging site and transmits time chunks via the ADIOS channels SSSSS_ECEI_NN. Here SSSSS denotes the shot number and NN enumerates the ADIOS channels. The* ***processor*** *runs at the HPC site, recieves the data and submits it for processing through a* `task_list`*. .*

```
21    workers = []
22    for _ in range(n_thr):
23       w = threading.Thread(target=consume,
24                            args=(dq, task_list))
25       w.start()
26       workers.append(w)
27
28
29    while True:
30      stepStatus = reader.BeginStep()
31      if stepStatus:
32        stream_data = a2_reader.Get(varname)
33        dq.put_nowait((stream_data,
34                       reader.CurrentStep()))
35        reader.EndStep()
36      else:
37        break
38
39    worker.join()
40    dq.join()
```

To access the many cores available, `processor` is launched as an MPI program under control of `mpi4py.futures`: `srun -n NP -m mpi4py.futures processor.py`. The `mpi4py` documentation suggests to run as `mpiexec -n 1 -usize NP processor.py` but unfortunately Cori's job system supports neither `mpiexec` nor defining the universe size by environment variables. The number of MPI ranks should be approximately equal to the workers requested in the PoolExecutors, `NP == NF + NA - 1`.

Then `a2_reader` is instantiated with `cfg[transport_rx]`, mirroring the configuration of the writer. After defining a queue for inter-process communication, a group of worker threads is started. In the main loop `a2_reader` consumes incoming time chunk data from the ADIOS stream and enqueues them. At the same time, the array of worker tasks dequeues time chunks data and passes it to the `task_list`.

The data analysis code is implemented by cython kernels which are described in a later subsection. While the low-level implementation of Eqs. (1) - (4) is in cython, `Delta` encapsulates them by the `task` class. Sans initialization the relevant class interface is implemented as

```
class task():
...
def calc_and_store(self, data, **kwargs):
  result = self.kernel(data, **kwargs)
  self.storage_backend.store(result, tidx)

def submit(self, executor, data, tidx):
  ...
  _ = [executor.submit(self.calc_and_store, data,
                       ch_it, tidx)
       for ch_it in self.get_dispatch_sequence()]
```

The call of an analysis kernel happens in `calc_and_store`. Once the kernel returns, the analyzed data is immediately stored. This allows us to submit a large number of analysis task in parallel in a `fire-and-forget` way. Implementing analysis and storage as separate functions would introduce dependencies between futures returned by `executor.submit`. Grouping analysis and storage together guarantees that once `calc_and_store` returns, the data has been analyzed and stored. In order to minimize data communication, `submit` launches `calc_and_store` for an

exhaustive list of channel pair combinations which is accessed by `get_dispatch_sequence()`.

Since the ECEI analysis tasks for the workflow at hand expects Fourier transformed data, the analysis kernels are called sequentially right after the Fourier transformed data becomes available. This logic is implemented by the `task_list` class:

```
from scipy.signal import stft

class task_list():

  def submit(self, data, tidx):
    fft_future = self.executor_fft.submit(stft,
                                          data,
                                          **kwargs)

    for task in self.task_list:
      task.submit(self.executor_anl,
                  fft_future.result(), tidx)
```

Executing the analysis tasks after the Fourier transformation further reduces interdependencies in the workflow, i.e. this implementation awaits only a single future. Without collecting the analysis tasks in a list one may for example execute Fourier transformations prior to launching each individual analysis kernel. This particular choice would increase the number of Fourier transformations by a factor of four and may seem like a poor choice. On the other hand would this result in less communication across the MPI ranks and may perform better in situations where communication between MPI ranks becomes a bottleneck.

Explored alternative architectures

`Delta` relies on the `futures` interface defined in PEP 3148 to launch data analysis kernels on an HPC resource [PEP3148]. Since both Cori and ADIOS are designed for MPI application we use the `mpi4py` [mpi4py] implementation. Being a standard interface, other implemenations like `concurrent.futures` can readily be used. The Python Standard Library defines the interface as `executor.submit(fn, *args **kwargs)`. `Delta` wraps PEP 3148 `submit` calls in wrapper methods of the `task` and `task_list` class in order to pass kernel-dependent keyword arguments and in order to facilitate more flexible launch configuration on multiple executors.

Besides `mpi4py` we explored executing `task.calc_and_store` calls on a `Dask` [dask] cluster. Exposing `concurrent.futures`-compatible interface, both libraries can be interchanged with little work. Running on a single node we found little difference in execution speed. However once the dask-distributed cluster was deployed on multiple nodes we observed a significant slowdown due to network traffic overhead. We did not investigate this problem any further.

As an alternative to using a queue with threads, we also explored using asynchronous I/O. In this scenario, the main task would define a coroutine receiving the data time chunks and a second one dispatching them to an executor. In our tested implementation, the coroutines would run in a main loop and communicate via a queue. Our experiments showed no measurable difference against a threaded implementation. On the other hand, the threaded implementation fits more naturally in the multiprocessing design approach.

Using data analysis codes `Delta`

In a broad sense, data analysis can be described as applying a transformation F to some data d,

$$y = F(d; \lambda_1, \ldots, \lambda_n), \tag{5}$$

given some parameters $\lambda_1 \ldots \lambda_n$. Translating the relation between the F and d into an object-oriented setting is not always straightforward and one needs to have the application in mind when designing a library. The approach taken by general-purpose packages such as `scipy` or `scikit-learn` is to implement a transformation F as a class and interface to data through its member functions. Taking Principal Component Analysis in scikit-learn as an example, the default way of applying it to data is

```
from sklearn.decomposition import PCA
X = np.array([...])
pca = PCA(n_components=2)
pca.fit_transform(X)
```

This approach has proven itself useful and is the common way of organizing libraries. `Delta` deviates slightly from this approach and calls transformations in the `calc_and_store` member function of the `task_ecei` class. The specific kernel to be called is configured in the objects initialization:

```
from kernels import kernel_crossphase, ...
class task():
   def __init__(self, cfg):
      ...
   if (cfg["analysis"] == "cross-phase"):
      self.kernel = kernel_crossphase
   elif (cfg["analysis"] == cross-power"):
      self.kernel = kernel.crosspower
   ...
  def calc_and_store(self, data, ...):
     ...
     result = self.kernel(data, ...)
```

At the time of writing, `Delta` only implements a workflow for ECEI data and this design choice minimizes the number of classes present in the framework. Grouping the data analysis methods by diagnostic also allows to execute diagnostic-specific pre-transformations that are best performed after transfer to the `processor` collectively. One may wish for example to distribute calculations of the 18336 channel pair combinations among multiple `task` instances. This approach lets us seamlessly do that. Once the requirements and use cases have stabilized we will explore suitable generalizations such as object factories for the `task_list` class.

In summary, the architecture of `Delta` implements data streaming using time-stepping interface of ADIOS and data analysis using PEP 3148 compatible executors. In order to increase performance we choose to use two PoolExecutors. The first executor is used to execute short Fourier Transformations of the the input data for the entire analysis task group. The second pool executor is available for running the analysis kernels and immediate storage of the results.

Performance analysis

While the overall performance of the framework can be measured by the walltime of the analysis workflow at hand, the complex composition of the framework requires us to understand the performance of its building blocks. Referring to figure 2, IO performance of the ADIOS library, the asynchronous receive-publish-submit strategy implemented by processor and finally the speed of individual analysis kernels contribute to the workflow walltime. Furthermore, the workflow walltime may be sensitive to the individual components interacting with one another. For example, even though the processor design aims to facilitate high-velocity data streams by using queues and multiple worker threads, a fast data stream ingested by the processor may negatively affect

the performance of the PoolExecutors by submitting too many tasks in a short time. It may well be that slower data streaming rate result in a smaller workflow walltime. Given these considerations we start be investigating the performance of individual components in this section and finally investigate the performance of the framework on the ECEI workflow.

Performance of the WAN connection

As a first step we measure the practically available bandwidth between the KSTAR and NERSC DTNs using the network performance tool iperf3 [iperf]. Multiple data streams are often necessary to exhaust high-bandwidth networks. Varying the number of senders from 1 to 8, we measure data transfer rates from 500 MByte/sec using 1 process up to a peak rate of 1500 MByte/sec using 8 processes, shown in Figure 3. Using 1 thread we find that the data transfer rate is approximately 500 MByte/sec with little variation throughout the benchmark. Running the 2 and 4 process benchmark we see initial transfer rates of more than 1000 MByte/sec. After about 5 to 8 seconds, TCP observes network congestion and falls back to fast recovery mode where the transfer rates increase to the approximately the initial transfer rates until the end of the benchmark run. The 8 process benchmark shows a qualitatively similar behaviour but the congestion avoidance starts at approximately 15 seconds where the transfer enters a fast recovery phase.

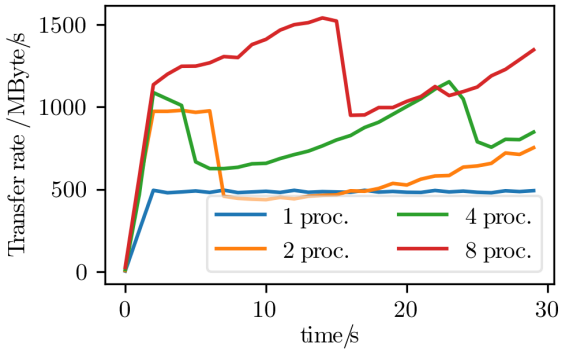


Fig. 3: *Data transfer rates between the KSTAR and NERSC DTNs measured using iperf3 using 1, 2, 4, and 8 processes*

While we measured the highest bandwidth when transferring with 8 process, `Delta` currently only implements single process data transfers.

Data Analysis Kernels

As seen in the code-example above, `Delta` implements data analysis routines as computational kernels. These are implemented in cython to circumvent the global interpreter lock and utilize multiple cores. Measuring the average execution time over 10 runs on a Cori compute node we find that the kernels demonstrate a strong scaling for up to 16 threads, shown in Fig. 4. Using more 32 threads results in sub-linear speedup.

Performance of the ECEI workflow

Having established the performance of the individual components we continue by benchmarking the performance of `Delta` performing the entire ECEI analysis workflow. The task at hand is to calculate Eqs.(1) - (4) for 18836 unique channel pair combinations per time chunk. Each time chunk consists of $s_{ch} = 10,000$ samples

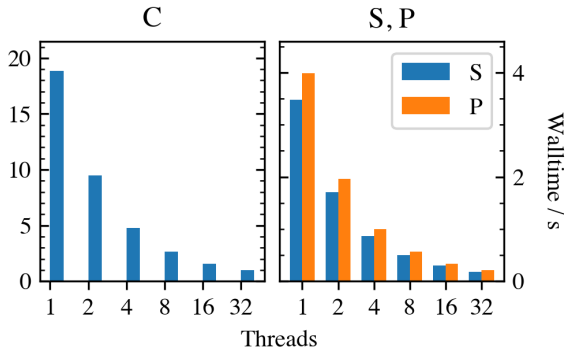


Fig. 4: *Runtime of the multi-threaded kernels for coherence C, cross-power S and cross-phase P compared against numpy implementations.*

Scenario	Walltime	N_{ch}	Bandwidth
file	347s	500	350 MByte/sec
2-node	358s	485	95 MByte/sec
3-node	339s	463	450 MByte/sec

TABLE 3: *Performance metrics for the ECEI workflow in the benchmarked scenarios.*

for 192 individual channels. A total of $N_{ch} = 500$ time chunks are to be processed, for a total of about 5 GByte of data.

The performance of `Delta` depends on the individual performance of multiple components, such as the data streaming velocity, lag introduced by using queue inter-process communication, spawning processes on the executors, MPI communication as well as their interplay with one another. Having benchmarked individual components in the previous section, we now benchmark the runtime of `Delta` performing the ECEI spectral analysis workflow in three scenarios. In the `file` scenario, the `processor` reads data from a local ADIOS file. No data is streamed. In the `2-node` scenario, data is streamed from the `generator` running on the NERSC DTN to Cori. In the `3-node` scenario, data is streamed from the KSTAR DTN to the NERSC DTN and forwared to Cori - this is the scenario shown in 2. Both the `2-` and `3-node` scenario use ADIOS DataMan engine for data streaming. All runs are performed on an allocation using 32 Cori nodes partitioned into 128 MPI ranks with 16 Threads each for a total of 2048 CPU cores.

Table 3 lists the Walltime and the number of proccessed time chunk N_{ch} and the utilized bandwidth. Walltime refers to the walltime as measured by the `processor` and N_{ch} gives the number of time chunks analyzed by the processor. The utilized bandwidth refers to the I/O speed achieved when reading from disk in the `file` scenario, the average data transfer rate from the NERSC DTN to Cori in the `2-node` scenario and as the average data transfer rate from the KSTAR DTN to the NERSC DTN in the `3-node scenario`.

The measured walltime for the file-based workflow is 352s, 358s for the 2-node scenario and 339s for the 3-node scenario. Only minor packet loss occurs using the current implementation of the DataMan engine. In order to mitigate packet loss the `generator` pauses a tenth of a second after sending any packet from the NERSC DTN to Cori, resulting in a bandwidth of 95

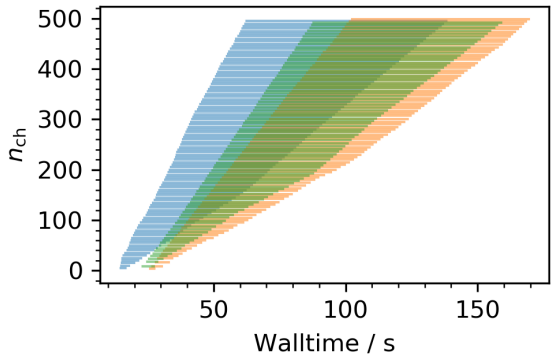


Fig. 5: *Horizontal bars mark the time that the a given time chunk n_{ch} spends in the queue of the processor. The color legend is shown in Figure 6*

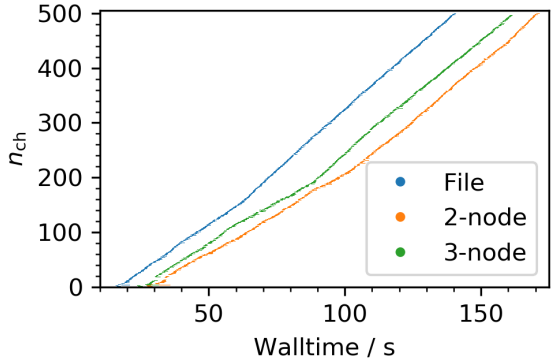


Fig. 6: *Horizontal bars mark the during which the STFT for each time chunk data is executed*

MByte/sec for the `2-node` scenario. In the `3-node` scenario we show that `Delta` can ingest high velocity data streams from KSTAR to NERSC and perform analysis on them. As in the `2-node` scenario, we limit the bandwidth from the NERSC DTN to Cori by pausing a fraction of a second before relaying a time chunk. On average, `Delta` performs the entire analysis workflow as fast in a streaming setting as it does when reading from the local file system. The average time to analyze a single time chunk is about 0.7 seconds, independent of the workflow.

Figure 5 shows the amount of time that data for a given time chunk, $n_{ch} = 1...N_{ch}$, spends in the queue of the processor. All three scenarios show a similar trend - the amount of time a time chunk spends in the queue increases with the time when it is enqueued. This suggests that data is streamed faster to the `processor` than the MPI ranks perform data analysis. This implies that the queue acts as a cache for the incoming time chunk data. Running the `file` scenario, the processor loads data almost immediatetly after it starts up. For the `2-node` and `3-node` scenarios the start time of the components on their respective machines is not coordinated. This causes the first time chunk data to arrive at varying times for the three scenarios.

As time chunks are dequeued, they are subject to a STFT. Figure 6 denotes the time where the STFT of each time chunk is performed with horizontal bars. The beginning of a horizontal bar indicates where the STFT with the time chunk data is submitted on `executor_fft` and the end of a bar marks the time STFT is finished. Common for all three scenarios is that the STFTs with the longest execution time are the ones for the first time chunks received. Also, the majority of the STFTs is executed in approximately one second. Equivalent STFT evaluations outside Delta take about 0.15s on Cori. On average the STFT when called from the streaming workflow is slower by a factor of 6. We believe that this long execution time is in part explained by MPI communication overhead.

Finally, Figures 7, 8 and 9 show the utilization of the MPI ranks over time. The MPI ranks execute the STFT and analysis kernels, the figures only show the time where analysis kernels are executed. All three scenarios show a low usage of available MPI ranks, approximately 16 - 20 in the beginning of the run. After all time chunks are dequeued and Fourier transformed, all available MPI ranks are used. Color encodes the different analysis kernels. For example, green bars show time at which a cross-correlation kernel is executed. The majority of the compuation

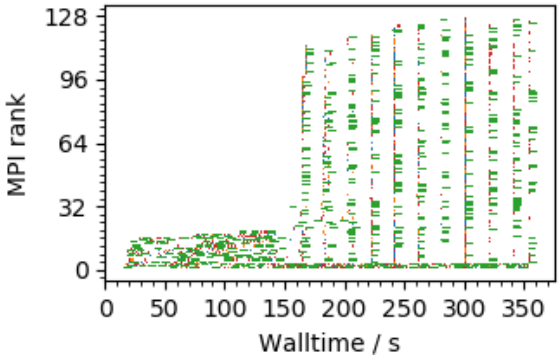


Fig. 7: *MPI rank utilization for the `file` scenario. Colored bars mark the execution time of analysis kernels. Blue bars denote cross-phase, orange bars denote cross-power, green bars denote cross-correlation and red bars denote coherence.*

time is consumed by cross-correlation kernels. This observation agrees with the performance analysis that showed that the cross-correlation kernel is the most time consuming.

Conclusions and future work

We demonstrate that `Delta` can facilitate near real-time analysis of high-velocity streaming data. In our experiments we achieved streaming rates of about 350 MByte/sec and execute a spectral analysis workflow on ECEI measurements in less than 4 minutes. Performing the analysis in the streaming scenario, illustrated

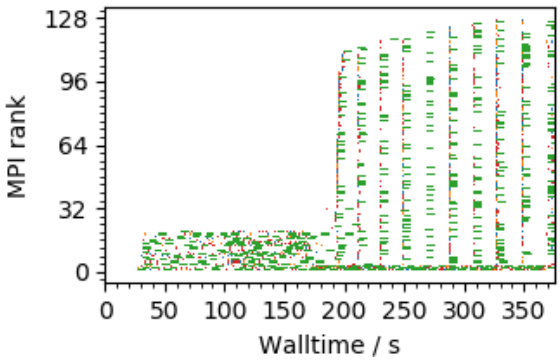


Fig. 8: *MPI rank utilization for the `2-node` scenario. The color encoding of the analysis kernels is the same as in Figure 7*

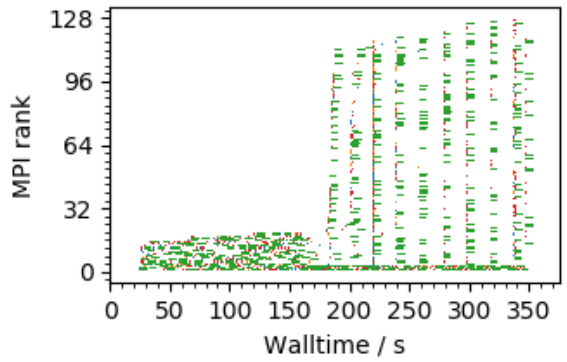


Fig. 9: *MPI rank utilization for the* `3-node` *scenario. The color encoding of the analysis kernels is the same as in Figure 7*

in Figure 1, comes with only a negligible performance impact as compared to using local filesystem IO. ADIOS manages to utilize about 70% of the available bandwidth for data streaming from KSTAR to NERSC in the streaming analysis workflow. `mpi4py` PoolExecutors facilitate an flexible execution of work items on Cori, as required for our workflow where data arrive at high velocity. Furthermore, python queues reliably facilitate inter-process communication and act as a data cache under the tested IO loads.

In the current form, there are multiple shortcomings of the framework that need to be addressed. Firstly, the DataMan engine received an experimental feature to mitigate packet loss. Secondly, implementation details of MPI on Cori limit us to effectively a single PoolExecutor. We are planning to investigate this more closely and aim to properly separate the execution space of the STFT and the analysis kernels. Thirdly, the framework will be generalized in order to facilitate more data analysis tasks. Finally, we are working on adapting `Delta` for next generation HPC facilities which heavily rely on graphical processing units to provide processing power.

Another issue we plan to address is to make `Delta` more adaptive. This includes developing machine learning algorithm for data compression and to decide which data batches are to be offloaded to HPC resources for in-depth analysius. For example, ECEI time chunk data that is not likely to be relevant for magnetic island studies could be analyzed with fast, coarse routines at a local workstation while relevant data could be forwarded to in-depth analysis routines.

Acknowledgements

The authors would like to acknowledge the excellent technical support from engineers and developers at the National Energy Research Scientific Computing Center. This work used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. `Delta` is available on github: [Git] All data used to generate the plots in this article can be accessed on Zenodo [Zen] .

References

[PPY] PlasmaPy Community, Nicholas A. Murphy, Andrew J. Leonard et al. PlasmaPy: an open source community-developed Python package for plasma physics. Zenodo. http://doi.org/10.5281/zenodo.1238132

[Men15] O. Meneghini, S.P. Smith, L.L. Lao et al. *Integrated modeling applications for tokamak experiments with OMFIT* Nucl. Fusion **55** 083008 (2015)

[Git] Ralph Kube (2020, June). DELTA-FUSION (aDaptive rEaL Time Analysis of big fusion data). Retrieved from https://github.com/rkube/delta

[Ent18] S. Entler, J. Horacek, T. Dlouhy and V. Dostal *Approximation of the economy of fusion energy* Energy 152 p. 489 (2018)

[KSTAR] G.S. Lee, J. Kim, S.M. Hwang et al. *The design of the KSTAR tokamak* Fus. Eng. Design 46 405-411 (1999) https://doi.org/10.1016/S0920-3796(99)00032-0

[Cos74] A.E Costley, R.J. Hastie, J.W.M. Paul, and J. Chamberlain *Electron Cyclotron Emission from a Tokamak Plasma: Experiment and Theory* Phys. Rev. Lett. 33 p. 758 (1974).

[Yun14] G.S. Yun, W. Lee, M.J. Choi et al. *Quasi 3D ECE imaging system for study of MHD instabilities in KSTAR* Rev. Sci. Instr. 85 11D820 (2014) http://dx.doi.org/10.1063/1.4890401

[Bal17] E.A. Baltz, E. Trask, M. Binderbauer et al. *Achievement of Sustained Net Plasma Heating in a Fusion Experiment with the Optometrist Algorithm* Sci. Reports 6425 (2017) https://doi.org/10.1038/s41598-017-06645-7

[Bel18] V. A. Belyakov and A. A. *Kavin Fundamentals of Magnetic Thermonuclear Reactor Design* Chapter 8 Woodhead Publishing Series in Energy

[Yun10] G. S. Yun, W. Lee, M. J. Choi et al. *Development of KSTAR ECE imaging system for measurement of temperature fluctuations and edge density fluctuations* Rev. Sci. Instr. 81 10D930 (2010) https://dx.doi.org/10.1063/1.3483209

[Cho17] M. J. Choi, J. Kim, J.-M. Kwon et al. *Multiscale interaction between a large scale magnetic island and small scale turbulence* Nucl. Fusion **57** 126058 (2017) https://doi.org/10.1088/1741-4326/aa86fe

[cori] National Energy Research Scientific Computing Center. Cori. Retrieved from https://docs.nersc.gov/systems/cori/

[top500] @top500supercomp (2019, Nov) We are proud to announce the 54th edition of the TOP500 list! China extends lead in number of TOP500 supercomputers, US holds on to performance advantage. To view the full list, visit https://top500.org/lists/2019/11/ Retrieved from https://twitter.com/top500supercomp/status/1196428698339160065

[dtn] Energy Sciences Network. Data Transfer Nodes. Retrieved from http://es.net/science-engagement/technical-and-consulting-services/data-transfer-nodes/

[Xie12] B. Xie, J. Chase, D. Dillow et al. *Characterizing output bottlenecks in a supercomputer* SC '12: Proceedings of the International conference on High Performance Computing, Networking, Storage and Analysis https://doi.org/10.1109/SC.2012.28

[nerscdtn] National Energy Research Scientific Computing Center. Data Transfer Nodes. Retrieved from https://docs.nersc.gov/systems/dtn/

[iperf] ESnet / Lawrence Berkeley National Laboratory (2014, July 7) iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr

[adios] Oak Ridge National Laboratory (2018, April 5) ADIOS 2: The Adaptable Input/Output System version 2. Retrieved from https://adios2.readthedocs.io/en/latest/index.html

[PEP3148] B. Quinlan *PEP 3148 futures - execute computations asynchronously* 2009 Retrieved from https://www.python.org/dev/peps/pep-3148/

[mpi4py] L. Dalcin, R. Paz and M. Storti *MPI for Python* Journal of Parallel and Distributed Computing, 65(9): 1108–1115, 2005 https://doi.org/10.1016/j.jpdc.2005.03.010

[dask] M. Rocklin *Dask: Parallel Computation with Blocked Algorithms and Task Scheduling* Proceedings of the 14th Python in Science Conference p.126-132 2015 DOI: 10.25080/Majora-7b98e3ed-013

[FFT] Heinzel, G., Rüdiger, A., & Schilling, R. (2002). Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows. http://hdl.handle.net/11858/00-001M-0000-0013-557A-5

[Zen] Kube, Ralph, Churchill, R Michael, Chang, CS, et al. (2020). Leading magnetic fusion energy science into the big-and-fast data lane. Zenodo http://doi.org/10.5281/zenodo.3871700

SHADOW: A workflow scheduling algorithm reference and testing framework

Ryan W. Bunney[§‡*], Andreas Wicenec[§‡], Mark Reynolds[‡]

Abstract—As the scale of science projects increase, so does the demand on computing infrastructures. The complexity of science processing pipelines, and the heterogeneity of the environments on which they are run, continues to increase; in order to deal with this, the algorithmic approaches to executing these applications must also be adapted and improved to deal with this increased complexity. An example of this is workflow scheduling, algorithms for which are continually being developed; however, in many systems that are used to deploy science workflows for major science projects, the same algorithms and heuristics are used for scheduling. We have developed SHADOW, a workflow-oriented scheduling algorithm framework built to address an absence of open implementations of these common algorithms, and to facilitate the development and testing of new algorithms against these 'industry standards'. SHADOW has implementations of common scheduling heuristics, with the intention of continually updating the framework with heuristics, metaheuristics, and mathematical optimisation approaches in the near future. In addition to the algorithm implementations, there is also a number of workflow and environment generation options, using the companion utility SHADOWGen; this has been provided to improve the productivity of algorithm developers in experimenting with their new algorithms over a large variety of workflows and computing environments. SHADOWGen also has a translation utilities that will convert from other formats, like the Pegasus DAX file, into the SHADOW-JSON configuration. SHADOW is open-source and uses key SciPy libraries; the intention is for the framework to become a reference implementation of scheduling algorithms, and provide algorithm designers an opportunity to develop and test their own algorithms with the framework. SHADOW code is hosted on GitHub at https://github.com/myxie/shadow; documentation for the project is available in the repository, as well as at https://shadowscheduling.readthedocs.org.

Introduction

To obtain useful results from the raw data produced by science experiments, a series of scripts or applications is often required to produce tangible results. These application pipelines are referred to as Science Workflows [ALRP16], which are typically a Directed-Acyclic Graph (DAG) representation of the dependency relationships between application tasks in a pipeline. An example of science workflow usage is Montage[1], which takes sky images and re-projects, background corrects and add astronomical images into custom mosaics of the sky [BCD+08], [JCD+13]. A Montage pipeline may consist of more than 10,000 jobs, perform more than 200GB of I/O (read and write), and take 5 hours to run [JCD+13]. This would be deployed using a workflow management system (for example, Pegasus [DVJ+15]), which coordinates the deployment and execution of the workflow. It is this workflow management system that passes the workflow to a workflow scheduling algorithm, which will pre-allocate the individual application tasks to nodes on the execution environment (e.g. a local grid or a cloud environment) in preparation for the workflow's execution.

The processing of Science Workflows is an example of the DAG-Task scheduling problem, a classic problem at the intersection of operations research and high performance computing [KA99a]. Science workflow scheduling is a field with varied contributions in algorithm development and optimisation, which address a number of different sub-problems within the field [WWT15], [CCAT14], [BÇRS13], [HDRD98], [RB16], [Bur]. Unfortunately, implementations of these contributions are difficult to find; for example, implementations that are only be found in code that uses it, such as in simulation frameworks like WorkflowSim [THW02], [CD12]; others are not implemented in any public way at all [YB06], [ANE10]. These are also typically used as benchmarking or stepping stones for new algorithms; for example, the Heterogeneous Earliest Finish Time (HEFT) heuristic continues to be used as the foundation for scheduling heuristics [DFP12], [CCCR18], meta-heuristics, and even mathematical optimisation procedures [BBL+16], despite being 20 years old. The lack of a consistent testing environment and implementation of algorithms makes it hard to reproduce and verify the results of published material, especially when a common workflow model cannot be verified.

Researchers benefit as a community from having open implementations of algorithms, as it improves reproducibility and accuracy of benchmarking and algorithmic analysis [CHI14]. There exists a number of open-source frameworks designed for testing and benchmarking of algorithms, demonstrate typical implementations, and provide an infrastructure for the development and testing of new algorithms; examples include NLOPT for non-linear optimisation in a number of languages (C/C++, Python, Java) [Joh], NetworkX for graph and network implementations in Python, MOEA for Java, and DEAP for distributed EAs in Python [DRFG+12]. SHADOW (Scheduling Algorithms for DAG Workflows) is our answer to the absence of Workflow Scheduling-based algorithm and testing framework, like those discussed above. It is an algorithm repository and testing environment, in which the performance of single- and multi-objective workflow scheduling algorithms may be compared to implementations of common algorithms. The intended audience of SHADOW is those

* *Corresponding author: ryan.bunney@research.uwa.edu.au*
§ *International Centre for Radio Astronomy Research*
‡ *University of Western Australia*

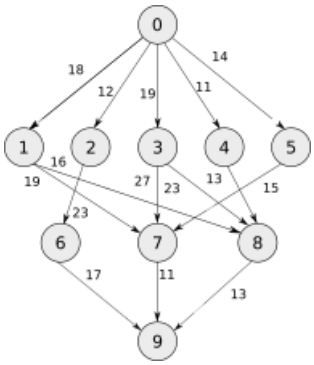


Fig. 1: *A sample DAG; vertices represent compute tasks, and edges show precedence relationships between nodes. Vertex- and edge-weights are conventionally used to describe computational and data costs, respectively. This is adapted from [THW02], and is a simple example of the DAG structure of a science workflow; a typical workflow in deployment will often be more complex and contain many hundreds of nodes and edges.*

developing and testing novel workflow scheduling algorithms, as well as those interested in exploring existing approaches within an accessible framework.

To the best of our knowledge, there is no single-source repository of implementations of DAG or Workflow scheduling algorithms. The emphasis in SHADOW is on reproducibility and accuracy in algorithm performance analysis, rather than a simulated demonstration of the application of a particular algorithm in certain environments. Additionally, with the popularity of Python in other domains that are also growing within the workflow community, such as Machine and Deep Learning, SHADOW provides a frictionless opportunity to integrate with the frameworks and libraries commonly used in those domains.

Workflow Scheduling

A workflow is commonly represented in the literature as a Directed Acyclic Graph (DAG) [CK88], [CA93], [Ull75], [KA99a]; a sequence of tasks will have precedence constraints that limit when a task may start. A DAG task-graph is represented formally as a graph $G = (V,E)$, where V is a set of v vertices and E is a set of e edges [KA99a]; an example is featured in Figure 1, which will be build upon as the paper progresses. Vertices and Edges represent communication and computation costs respectively. The objective of the DAG-scheduling problem is to map tasks to a set of resources in an order and combination that minimise the execution length of the final schedule; this is referred to as the *makespan*.

The complexity and size of data products from modern science projects necessitates dedicated infrastructure for compute, in a way that requires re-organisation of existing tasks and processes. As a result, it is often not enough to run a sequence of tasks in series, or submit them to batch processing; this would likely be computationally inefficient, as well taking as much longer than necessary. As a result, science projects that have computationally- and data-intensive programs, that are interrelated, have adopted the DAG-scheduling model for representing their compute pipelines; this is where science workflow scheduling is derived.

Design and Core Architecture

Design

SHADOW adopts a workflow-oriented design approach, where workflows are at the centre of all decisions made within the framework; environments are assigned to workflows, algorithms operate on workflows, and the main object that is manipulated and interacted with when developing an algorithm is likely to be a workflow object.

By adopting a workflow-oriented model to developing algorithms to test, three important outcomes are achieved:

- Freedom of implementation; for users wishing to develop their own algorithms, there is no prohibition of additional libraries or data-structures, provided the workflow structure is used within the algorithm.
- Focus on the workflow and reproducibility; when running analysis and benchmarking experiments, the same workflow model is used by all algorithms, which ensures comparisons between differing approaches (e.g. a single-objective model such as HEFT vs. a dynamic implementation of a multi-objective heuristic model) are applied to the same workflow.
- Examples: We have implemented popular and well-documented algorithms that are commonly used to benchmark new algorithms and approaches. There is no need to follow the approaches taken by these implementations, but they provide a useful starting point for those interested in developing their own.

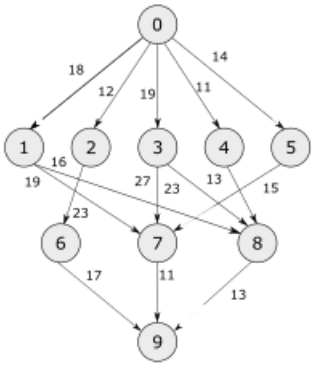
SHADOW is not intended to accurately simulate the execution of a workflow in an real-world environment; for example, working with delays in processing, or node failure in a cluster. Strategies to mitigate these are often implemented secondary to the scheduling algorithms, especially in the case of static scheduling, and would not be a fair approach to benchmarking the relative performance between each application. Instead, it provides algorithms that may be used, statically or dynamically, in a larger simulation environment, where one would be able to compare the specific environmental performance of one algorithm over another.

Architecture

SHADOW is split into three main components that are separated by their intended use case, whether it be designing new algorithms, or to benchmark against the existing implementations. These components are:

- `models`
- `algorithms`
- `visualiser`

The `models` module is likely the main entry point for researchers or developers of algorithms; it contains a number of key components of the framework, the uses of which are demonstrated both in the `examples` directory, as well as the implemented sample algorithms in the `algorithms` module. The `algorithms` module is concerned with the implementations of algorithms, with the intention of providing both a recipe for implementing algorithms using SHADOW components, and benchmark implementations for performance analysis and testing. The visualiser is a useful way to add graphical components to a benchmarking recipe, or can be invoked using the command line interface to quickly run one of the in-built algorithms.



	FLOPs
0	119000
1	92000
2	95000
3	109000
4	107000
5	169000
6	110000
7	147000
8	146000
9	101000

	FLOP/s
M1	7
M2	6
M3	11

Fig. 2: *An example workflow DAG adapted from [THW02] (the same workflow as in Figure 1); weights on the edges describe data products from the respective parent node being sent to the child. In SHADOW, task computation cost is represented by the total number of Floating Point Operations required to run the task (see Table 1). This is intended to alleviate the difficulty of converting the run-time between different test environment configurations.*

Workflow and Costs		**Environment**	
Task	*FLOPs*	*Machine*	*FLOP/s*
0	119000	`cat0_m0`	7000
1	92000	`cat1_m1`	6000
2	95000	`cat2_m2`	11000
3	109000		
4	107000		
5	169000		
6	110000		
7	147000		
8	146000		
9	101000		

TABLE 1: *Table of Task (Giga) FLOP requirements, with the (Giga) FLOP/second provided by each respective machine. It is intended to be applied to Figure 2.*

These components are all contained within the main `shadow` directory; there are also additional codes that are located in `utils`, which are covered in the **Additional Tools** section.

Models

The `models` module provides the `Workflow` class, the foundational data structure of shadow. Currently, a `Workflow` object is initialised using a JSON configuration file that represents the underlying DAG structure of the workflow, along with storing different attributes for task-nodes and edges in Figure 2 (which is an extension of Figure 1).

These attributes are implicitly defined within the configuration file; for example, if the task graph has compute demand (as total number of FLOPs/task) but not memory demand (as average GB/task), then the Workflow object is initialised without memory, requiring no additional input from the developer.

Using the example workflow shown in Figures 1 and 2, we can demonstrate how to initialise a `Workflow` in SHADOW, and what options exist for extending or adapting the object.

```
from shadow.models.workflow import Workflow
HEFTWorkflow = Workflow('heft.json')
```

The `heft.json` file contains the graph structure, based the JSON dump received when using networks. Nodes and their respective costs (computation, memory, monetary etc.) are stored with their IDs.

```
...
"nodes": [
    {
        "comp": 119000,
        "id": 0
    },
    {
        "comp": 92000,
        "id": 1
    },
    {
        "comp": 95000,
        "id": 2
    },
    ...
],
```

It is clear from Figure HEFT Edges in the graph, which are the dependency relationship between tasks, are described by links, along with the related data-products:

```
"links": [
    {
        "data_size": 18,
        "source": 0,
        "target": 1
    },
    {
        "data_size": 12,
        "source": 0,
        "target": 2
    },
    ...
```

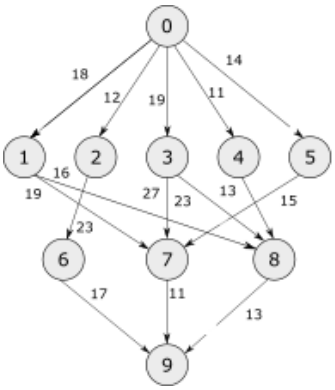
For example, looking at Figure 2 we see the dependency between tasks 0 and 1, and the weight 18 on the edge. This is reflected in the above component of the JSON file.

NetworkX is used to form the base-graph structure for the workflow; it allows the user to specify nodes as Python objects, so tasks are stored using the SHADOW `Task` object structure. By using the `NetworkX.DiGraph` as the storage object for the workflow structure, users familiar with NetworkX may use with the SHADOW `Workflow` object in any way they would normally interact with a NetworkX `Graph`.

In addition to the JSON configuration for the workflow DAG, a Workflow object also requires an `Environment` object. `Environment` objects represent the compute platform on which the Workflow is executed; they are add to `Workflow` objects in the event that different environments are being analysed. The environment is also specified in JSON; currently, there is no prescribed way to specify an environment in code, although it is possible to do so if using JSON is not an option.

In our example, we have three machines on which we are attempting to schedule the workflow from Figure 2. The different performance of each machine is described in Table 1, with the JSON equivalent below:

```
"system": {
  "resources": {
    "cat0_m0": {
      "flops": 7000.0
      "mem":
      "io" :
    },
    "cat1_m1": {
      "flops": 6000.0
    },
```



	M1	M2	M3
0	14	16	9
1	12	19	18
2	11	13	19
3	13	8	17
4	12	13	10
5	13	16	9
6	7	15	11
7	5	11	14
8	18	12	20
9	21	7	16

Fig. 3: *This is a replication of the costs provided in [THW02]. The table shows a different run-time for each task-machine pairing. It is the same structure as Figure 2; however, the JSON specification is different to cater for the pre-calculated run-time on separate machines.*

```
    "cat2_m2": {
      "flops": 11000.0
    }
  },
  "rates": {
    "cat0": 1.0, # GB/s
    "cat1": 1.0,
    "cat2": 1.0
  }
}
```

Environments are added to the `Workflow` object in the following manner:

```
from shadow.models.environment import Environment
env = Environment('sys.json')
HEFTWorkflow.add_environment(env)
```

The Workflow class calculates task run-time and other values based on its current environment when the environment is passed to the Workflow); however, users of the environment class may interact with these compute values if necessary. Configuration files may be generated in a number of ways, following a variety of specifications, using the SHADOWGen utility.

It is also possible to use pre-calculated costs (i.e. completion time in seconds) when scheduling with SHADOW.

This approach is less flexible for scheduling workflows, but is a common approach used in the scheduling algorithm literature [KA99a], [KA99b], [**?**], [BM08], [YB06]; an example of this is shown in Figure 3. This can be achieved by adding a list of costs-per-tasks to the workflow specification JSON file, in addition to the following header. For example, if instead of the total FLOPS we had provided to us in Table 1, we instead had timed the run-time of the applications on each machine separately, the JSON for Figure 2 would reflect the following:

```
{
    "header" : {
    "time": true
    },
    ...

    "nodes": [
    {
        "comp": [
            14,
            16,
            9
        ],
        "id": 0
    },
    ...
}
```

The final class that may be of interest to algorithm developers is the `Solution` class. For single-objective heuristics like HEFT or min-min, the final result is a single solution, which is a set of machine-task pairs. However, for population- and search-based metaheuristics, multiple solutions must be generated, and then evaluated, often for two or more (competing) objectives. These solutions also need to be sanity-checked in order to ensure that randomly generated task-machine pairs still follow the precedence constraints defined by the original workflow DAG. The `Solution` provides a basic object structure that stores machines and task pairs as a dictionary of `Allocations`; allocations store the task-ID and its start and finish time on the machine. This provides an additional ease-of-use functionality for developers, who can interact with allocations using intuitive attributes (rather than navigating a dictionary of stored keywords). The `Solution` currently stores a single objective (makespan) but can be expanded to include other, algorithm-specific requirements. For example, NSGAII* ranks each generated solution using the non-dominated rank and crowding distance operator; as a result, the SHADOW implementation creates a class, `NSGASolution`, that inherits the basic `Solution` class and adds the these additional attributes. This reduces the complexity of the global solution class whilst providing the flexibility for designers to create more elaborate solutions (and algorithms).

Algorithms

These algorithms may be extended by others, or used when running comparisons and benchmarking. The `examples` directory gives you an overview of recipes that one can follow to use the algorithms to perform benchmarking.

The SHADOW approach to describing an algorithm presents the algorithm as a single entity (e.g. heft()), with an initialised workflow object passed as a function parameter. The typical structure of a SHADOW algorithm function is as follows:

- The main algorithm (the function to which a Workflow well be passed) is titled using its publication name or title (e.g. HEFT, PCP, NSGAII* etc.). Following PEP8, this is (ideally) in lower-case.
- Within the main algorithm function, effort has been made to keep it structured in a similar way to the pseudo-code as presented in the respective paper. For example, HEFT has two main components to the algorithm; Upward Ranking of tasks in the workflow, and the Insertion Policy allocation scheme. This is presented in SHADOW as:

```
def heft(workflow):
  """
  Implementation of the original 1999 HEFT algorithm.

  :params workflow: The workflow object to schedule
  :returns: The solution object from the scheduled workflow
  """
  upward_rank(workflow)
  workflow.sort_tasks('rank')
  insertion_policy(workflow)
  return workflow.solution
```

Complete information of the final schedule is stored in the `HEFTWorkflow.solution` object, which provides additional information, such as task-machine allocation pairs. It is convention

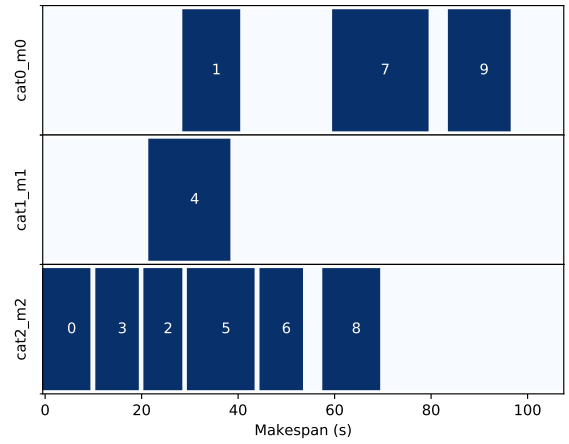


Fig. 4: *Result of running* `shadow.heuristic.heft` *on the graph shown in Figure 2. Final makespan is 98; gaps between tasks are indicative of data transfer times between parent and child tasks on different machines. This is generated using the* `AllocationPlot` *wrapper from the* `Visualiser`*.*

in SHADOW to have the algorithm return the Solution object attached to the workflow:

```
solution = heft(HEFTWorkflow)
```

In keeping with the generic requirements of DAG-based scheduling algorithms, the base Solution class prioritises makespan over other objectives; however, this may be amended (or even ignored) for other approaches. For example, the NSGAII algorithm uses a sub-class for this purpose, as it generates multiple solutions before ranking each solution using the crowded distance or non-dominated sort [SD94]:

```
class NSGASolution(Solution):
  """ A simple class to store each solutions'
      related information
  """

  def __init__(self, machines):
    super().__init__(machines)
    self.dom_counter = 0
    self.nondom_rank = -1
    self.crowding_dist = -1
    self.solution_cost = 0
```

Visualiser

SHADOW provides wrappers to `matplotlib` that are structured around the `Workflow` and `Solution` classes. The `Visualiser` uses the `Solution` class to retrieve allocation data, and generates a plot based on that information. For example, Figure 4 is the result of visualising the `HEFTWorkflow` example mentioned previously:

This can be achieved by creating a script using the algorithms as described above, and then passing the scheduled workflow to one of the Visualiser classes:

```
from shadow.visualiser.visualiser import AllocationPlot

sample_allocation = AllocationPlot(
    solution=HEFTWorkflow.solution
)

sample_allocation.plot(
    save=True,
    figname='sample_allocation.pdf'
)
```

Additional tools

Command-line interface

SHADOW provides a simple command-line interface (CLI) that allows users to run algorithms on workflows without composing a separate Python file to do so; this provides more flexibility and allows users to use a scripting language of their choice to run experiments and analysis.

```
python3 shadow.py algorithm heft \
'heft.json' 'sys.json'
```

It is also possible to use the `unittest` module from the script to run through all tests if necessary:

```
python3 shadow.py test --all
```

SHADOWGen

SHADOWGen is a utility built into the framework to generate workflows that are reproducible and interpretable. It is designed to generate a variety of workflows that have been documented and characterised in the literature in a way that augments current techniques, rather than replacing them entirely.

This includes the following:

- Python code that runs the GGen graph generator[2], which produces graphs in a variety of shapes and sizes based on provided parameters. This was originally designed to produce task graphs to test the performance of DAG scheduling algorithms.
- DAX Translator: This takes the commonly used Directed Acyclic XML (DAX) file format, used to generate graphs for Pegasus, and translates them into the SHADOW format. Future work will also interface with the Workflow-Generator code that is based on the work conduced in [BCD+08], which generates DAX graphs.
- DALiuGE/EAGLE Translator [WTV+17]: EAGLE logical graphs must be unrolled into Physical Graph Templates (PGT) before they are in a DAG that can be scheduled in SHADOW. SHADOWGen will run the DALiUGE unroll code, and then convert this PGT into a SHADOW-based JSON workflow.

Cost generation in SHADOWGen

A majority of work published in workflow scheduling will use workflows generated using the approach laid out in [BCD+08]. The five workflows described in the paper (Montage, CyberShake, Epigenomics, SIPHT and LIGO) had their task run-time, memory and I/O rates profiled, from which they created a WorkflowGenerator tool[3]. This tool uses the distribution sizes for run-time etc., without requiring any information on the hardware on which the workflows are being scheduled. This means that the characterisation is only accurate for that particular hardware, if those values are to be used across the board; testing on heterogeneous systems, for example, is not possible unless the values are to be changed.

This is dealt with in varied ways across the literature. For example, [RB18] use the distributions from [BCD+08] paper, and change the units from seconds to MIPS, rather than doing a conversion between the two. Others use the values taken from distribution and workflow generator, without explaining how

		Run-time		I/O Read		I/O Write		Peak Memory		CPU Util	
Job	Count	Mean (s)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (%)	Std. Dev
mProjectPP	2102	1.73	0.09	2.05	0.07	8.09	0.31	11.81	0.32	86.96	0.03
mDiffFit	6172	0.66	0.56	16.56	0.53	0.64	0.46	5.76	0.67	28.39	0.16
mConcatFit	1	143.26	0.00	1.95	0.00	1.22	0.00	8.13	0.00	53.17	0.00
mBgModel	1	384.49	0.00	1.56	0.00	0.10	0.00	13.64	0.00	99.89	0.00
mBackground	2102	1.72	0.65	8.36	0.34	8.09	0.31	16.19	0.32	8.46	0.10
mImgtbl	17	2.78	1.37	1.55	0.38	0.12	0.03	8.06	0.34	3.48	0.03
mAdd	17	282.37	137.93	1102.57	302.84	775.45	196.44	16.04	1.75	8.48	0.11
mShrink	16	66.10	46.37	411.50	7.09	0.49	0.01	4.62	0.03	2.30	0.03
mJPEG	1	0.64	0.00	25.33	0.00	0.39	0.00	3.96	0.00	77.14	0.00

TABLE 2: *Example profile of Montage workflow, as presented in [JCD+13]*

their run-time differ between resources [ANE13], [MJDN15]; Malawski et al. generate different workflow instances. using parameters and task run-time distributions from real workflow traces, but do not provide these parameters [MJDN15]. Recent research from [WLZ+19] still uses the workflows identified in [BCD+08], [JCD+13], but only the structure of the workflows is assessed, replacing the tasks from the original with other, unrelated examples.

SHADOWGen differs from the literature by using a normalised-cost approach, in which the values calculated for the run-time, memory, and I/O for each tasks is derived from the normalised size as profiled in [JCD+13] and [BCD+08]. This way, the costs per-workflow are indicative of the relative length and complexity of each task, and are more likely to transpose across different hardware configurations than using the varied approaches in the literature.

$$X' = \frac{(X \times n_{task}) - X_{min}}{X_{max} - X_{min}} \quad (1)$$

The distribution of values is derived from a table of normalised values using a variation on min-max feature scaling for each mean or standard deviation column in Table 2. The formula to calculate each task's normalised values is described in Equation 1; the results of applying this to Table 2 is shown in Table 3:

This approach allows algorithm designers and testers to describe what units they are interested in (e.g. seconds, MIPS, or FLOP seconds for run-time, MB or GB for Memory etc.) whilst still retaining the relative costs of that task within the workflow. In the example of Table 3, it is clear that mAdd and mBackground are still the longest running and I/O intensive tasks, making the units less of a concern.

Alternatives to SHADOW

It should be noted that existing work already addresses testing workflow scheduling algorithms in real-world environments; tools like SimGrid [CLQ], BatSim [DMPR17], GridSim [BM02], and its extensions, CloudSim [CRB+11] and WorkflowSim [CD12], all feature strongly in the literature. These are excellent resources for determining the effectiveness of the implementations at the application level; however, they do not provide a standardised repository of existing algorithms, or a template workflow model that can be used to ensure consistency across performance testing. Current implementations of workflow scheduling algorithms may be found in a number of different environments; for example, HEFT and dynamic-HEFT implementations exist in WorkflowSim[4] , but one must traverse large repositories in order to reach them. There are also a number of implementations that are present on open-source repositories such as GitHub, but these are not always official releases from papers, and it is difficult to keep track of multiple implementations to ensure quality and consistency. The algorithms that form the `algorithms` module in SHADOW are open and continually updated, and share a consistent workflow model. Kwok and Ahmed [KA99a] provide a comprehensive overview of the metrics and foundations of what is required when benchmarking DAG-scheduling algorithms, Maurya et al. maurya2018' extend this work and describe key features of a potential framework for scheduling algorithms; SHADOW takes inspiration from, and extends, both approaches.

Conclusion

SHADOW is a development framework that addresses the absence of a repository of workflow scheduling algorithms, which is important for benchmarking and reproducibility [MT18]. This repository continues to be updated, providing a resource for future developers. SHADOWGen extends on existing research from both the task- and workflow-scheduling communities in graph generation by using existing techniques and wrapping them into a simple and flexible utility. The adoption of a JSON data format compliments the move towards JSON as a standardised way of representing workflows, as demonstrated by the Common Workflow Language [CCH+16] and WorkflowHub[5].

Future work

Moving forward, heuristics and metaheuristics will continue to be added to the SHADOW algorithms module to facilitate broader benchmarking and to provide a living repository of workflow scheduling algorithms. Further investigation into workflow visualisation techniques will also be conducted. There are plans to develop a tool that uses the specifications in `hpconfig`[6], a Python class-based of different hardware (e.g. `class XeonPhi`) and High Performance Computing facilities (e.g `class PawseyGalaxy`). The motivation behind `hpconfig` is that classes can be quickly unwrapped into a large cluster or system, without having large JSON files in the repository or on disk; they also improve readability, as specification data is represented clearly as class attributes.

1. https://github.com/pegasus-isi/montage-workflow-v2
2. https://github.com/WorkflowSim/WorkflowSim-1.0/tree/master/sources/org/workflowsim/planning
3. https://github.com/perarnau/ggen
4. https://github.com/pegasus-isi/WorkflowGenerator
5. github.com/myxie/hpconfig
6. https://workflowhub.org/simulator.html

	Run-time		I/O Read		I/O Write		Peak Memory		CPU Util	
job	Mean (s)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (MB)	Std. Dev.	Mean (%)	Std. Dev
mProject PP	9.47	0.49	11.22	0.38	44.30	1.70	64.66	1.75	476.20	0.16
mDiffFit	10.61	9.00	266.27	8.52	10.29	7.40	92.61	10.77	456.48	2.57
mConcatFit	0.37	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.13	0.00
mBgModel	1.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.25	0.00
mBackground	9.42	3.56	45.78	1.86	44.30	1.70	88.65	1.75	46.32	0.55
mImgtbl	0.12	0.06	0.06	0.02	0.01	0.00	0.35	0.02	0.15	0.00
mAdd	12.50	6.11	48.83	13.41	34.34	8.70	0.70	0.08	0.37	0.00
mShrink	2.75	1.93	17.15	0.30	0.02	0.00	0.18	0.00	0.09	0.00
mJPEG	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.19	0.00

TABLE 3: *Updated relative cost values using the min-max feature scaling method described in Equation 1.*

REFERENCES

[ALRP16] Ehab Nabiel Alkhanak, Sai Peck Lee, Reza Rezaei, and Reza Meimandi Parizi. Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. *Journal of Systems and Software*, 113:1–26, March 2016. doi:10.1016/j.jss.2015.11.023.

[ANE10] S. Abrishami, M. Naghibzadeh, and D. Epema. Cost-driven scheduling of grid workflows using Partial Critical Paths. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 81–88, October 2010. doi:10.1109/GRID.2010.5697955.

[ANE13] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds. *Future Generation Computer Systems*, 29(1):158–169, January 2013. doi:10.1016/j.future.2012.05.004.

[BBL+16] T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. A Constraint Programming Scheduler for Heterogeneous High-Performance Computing Machines. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2781–2794, October 2016. doi:10.1109/TPDS.2016.2516997.

[BCD+08] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, November 2008. doi:10.1109/WORKS.2008.4723958.

[BÇRS13] Anne Benoit, Ümit V. Çatalyürek, Yves Robert, and Erik Saule. A Survey of Pipelined Workflow Scheduling: Models and Algorithms. *ACM Comput. Surv.*, 45(4):50:1–50:36, August 2013. doi:10.1145/2501654.2501664.

[BM02] Rajkumar Buyya and Manzur Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, November 2002. doi:10.1002/cpe.710.

[BM08] Jorge Barbosa and António P. Monteiro. A List Scheduling Algorithm for Scheduling Multi-user Jobs on Clusters. In José M. Laginha M. Palma, Patrick R. Amestoy, Michel Daydé, Marta Mattoso, and João Correia Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2008*, volume 5336, pages 123–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-92859-1_13.

[Bur] Andrew Marc Burkimsher. Fair, Responsive Scheduling of Engineering Workflows on Computing Grids. page 238.

[CA93] V. Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, March 1993. doi:10.1109/71.210815.

[CCAT14] Tarek Chaari, Sondes Chaabane, Nassima Aissani, and Damien Trentesaux. Scheduling under uncertainty: Survey and research directions. In *2014 International Conference on Advanced Logistics and Transport (ICALT)*, pages 229–234, May 2014. doi:10.1109/ICAdLT.2014.6866316.

[CCCR18] Y. Caniou, E. Caron, A. K. W. Chang, and Y. Robert. Budget-Aware Scheduling Algorithms for Scientific Workflows with Stochastic Task Weights on Heterogeneous IaaS Cloud Platforms. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 15–26, May 2018. doi:10.1109/IPDPSW.2018.00014.

[CCH+16] Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. *Common Workflow Language, v1.0*. figshare, United States, July 2016. doi:10.6084/m9.figshare.3115156.v2.

[CD12] Weiwei Chen and Ewa Deelman. WorkflowSim: A toolkit for simulating scientific workflows in distributed environments. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8, October 2012. doi:10.1109/eScience.2012.6404430.

[CHI14] Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. "Can I Implement Your Algorithm?": A Model for Reproducible Research Software. *arXiv:1407.5981 [cs]*, September 2014. arXiv:1407.5981.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988. doi:10.1109/32.4634.

[CLQ] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. page 7.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, January 2011. doi:10.1002/spe.995.

[DFP12] J. J. Durillo, H. M. Fard, and R. Prodan. MOHEFT: A multi-objective list-based method for workflow scheduling. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 185–192, December 2012. doi:10.1109/CloudCom.2012.6427573.

[DMPR17] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator. In Narayan Desai and Walfredo Cirne, editors, *Job Scheduling Strategies for Parallel Processing*, volume 10353, pages 178–197. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-61756-5_10.

[DRFG+12] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: A python framework for evolutionary algorithms. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference Companion - GECCO Companion '12*, page 85, Philadelphia, Pennsylvania, USA, 2012. ACM Press. doi:10.1145/2330784.2330799.

[DVJ+15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, May 2015. doi:10.1016/j.future.2014.10.008.

[HDRD98] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4):279–302, April 1998. doi:10.1016/S0305-0548(97)00055-5.

[JCD+13] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling

scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, March 2013. doi:10.1016/j.future.2012.08.015.

[Joh] Steven G. Johnson. The NLopt nonlinear-optimization package,.

[KA99a] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999. doi:10.1006/jpdc.1999.1578.

[KA99b] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999. doi:10.1145/344588.344618.

[MJDN15] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems*, 48:1–18, July 2015. doi:10.1016/j.future.2015.01.004.

[MT18] Ashish Kumar Maurya and Anil Kumar Tripathi. On benchmarking task scheduling algorithms for heterogeneous computing systems. *The Journal of Supercomputing*, 74(7):3039–3070, July 2018. doi:10.1007/s11227-018-2355-0.

[RB16] Maria Alejandra Rodriguez and Rajkumar Buyya. A taxonomy and survey on scheduling algorithms for scientific workflows in IaaS cloud computing environments. *Concurrency and Computation: Practice and Experience*, 29(8):e4041, 2016. doi:10.1002/cpe.4041.

[RB18] Maria A. Rodriguez and Rajkumar Buyya. Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms. *Future Generation Computer Systems*, 79:739–750, February 2018. doi:10.1016/j.future.2017.05.009.

[SD94] N. Srinivas and Kalyanmoy Deb. Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evol. Comput.*, 2(3):221–248, September 1994. doi:10.1162/evco.1994.2.3.221.

[THW02] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002. doi:10.1109/71.993206.

[Ull75] J. D. Ullman. NP-complete Scheduling Problems. *J. Comput. Syst. Sci.*, 10(3):384–393, June 1975. doi:10.1016/S0022-0000(75)80008-0.

[WLZ+19] Yuandou Wang, Hang Liu, Wanbo Zheng, Yunni Xia, Yawen Li, Peng Chen, Kunyin Guo, and Hong Xie. Multi-Objective Workflow Scheduling With Deep-Q-Network-Based Multi-Agent Reinforcement Learning. *IEEE Access*, 7:39974–39982, 2019. doi:10.1109/ACCESS.2019.2902846.

[WTV+17] C. Wu, R. Tobar, K. Vinsen, A. Wicenec, D. Pallot, B. Lao, R. Wang, T. An, M. Boulton, I. Cooper, R. Dodson, M. Dolensky, Y. Mei, and F. Wang. DALiuGE: A graph execution framework for harnessing the astronomical data deluge. *Astronomy and Computing*, 20:1–15, July 2017. doi:10.1016/j.ascom.2017.03.007.

[WWT15] Fuhui Wu, Qingbo Wu, and Yusong Tan. Workflow scheduling in cloud: A survey. *The Journal of Supercomputing*, 71(9):3373–3418, September 2015. doi:10.1007/s11227-015-1438-4.

[YB06] Jia Yu and Rajkumar Buyya. Scheduling Scientific Workflow Applications with Deadline and Budget Constraints Using Genetic Algorithms. https://www.hindawi.com/journals/sp/2006/271608/abs/, 2006. doi:10.1155/2006/271608.

Software Engineering as Research Method: Aligning Roles in Econ-ARK

Sebastian Benthall[§‡*], Mridul Seth[‡]

https://youtu.be/nxXr0LNdQUU

Abstract—While general purpose scientific software has enjoyed great success in industry and academia, domain specific scientific software has not yet become well-established in many disciplines where it has potential. Based on a survey of the literature as well as the authors' experiences contributing to Econ-ARK, a structural modeling toolkit for Economics, we argue that this is due to the well-documented skills gap that prevents researchers, publishers, and professors from making the most of the opportunities afforded by scientific software. When researchers professionalize their code, it enables more cumulative progress in research and facilitates technology transfer. When publishers release interactive computational artifacts, it enables constructionist learning of the material. When students are trained in software engineering, they can participate fully in the reproduction of their scientific field. This is especially the case for fields where scientific knowledge is represented in software code, as in the case of Economics. The skills gap will not be closed until software engineering is considered a core skill for the discipline. Software engineering should be reconceived as a research method.

Index Terms—computational method, computional thinking, constructionist learning, research software engineering

Computing in Education and Science

Ever since [Pap82] introduced constructionist learning using computers, educators have been enticed by the possibility that students could learn valuable knowledge by playing with software. While originally used as a tool for teaching mathematics, it was not long before Papert's Logo tool was also used in scientific education, teaching students not just about the abstract mathematical sphere, but about the physical world [ROP90]. The legacy of Logo is alive and well in NetLogo [TW04], which is used by both students and researchers alike in the study of complex and agent-based systems, and in the Python agent-based modeling (ABM) toolkit Mesa [DMJK15].[1]

Since, the ubiquity of computing and its increasingly central role in industry has prompted the spread of ideas that were once specific to computer science into other disciplines. [Win06] coined the term "computational thinking" for the general skills of managing abstraction, modularity, scalability, and robustness of systems. Now it refers to the cross-disciplinary use of these computational concepts [Guz08] [SGB13]. The question raised by computational thinking is how much computer science education is necessary for these cross-disciplinary uses of computation. Logo, after all, not only introduced students to mathematics, but also programming. But did it teach computational thinking?

The industrial demand for students educated in handling "Big Data" systems has since prompted a generalization of statistics beyond its discipline in a way that's analogous to the generalization of computer science. [Jor16] discusses this new industry demand for "inferential thinking". Together, computational thinking and inferential thinking have been reimagined by some as the foundation for a new form of cross-disciplinary data science curriculum [AD17] [EVDSLB19]. A key technological feature of these new curricula are digital notebooks that enable users to compose computational narratives that make computing more cognitively digestible to humans [PG15]. Now, Jupyter notebooks are widely used for collaboration on research and, in some places, as part of pedagogy.

Open source scientific software development has benefited from the influx of capital due to industry interest in data science applications. Software packages such as Numpy [WCV11], Pandas [McK11], and Scikit-learn [PVG+11] have become popular as industrial tools. At the same time, these tools have provided a foundation and aspirational example for more domain specific scientific libraries, such as astropy [RTG+13], Biopython [CAC+09], PsychoPy [Pei07], and SunPy [MPSC+13]. Scientific educators continue to see potential in the use of these tools to support the education of their students not only *about computation*, but *about the world* [Bar16], in a return to Papert's constructionist paradigm.

This vision of scientific research and education supported by open source domain specific scientific libraries faces two significant obstacles. The first is the development and sustainability of the software itself. Open source software projects in general are not guaranteed to succeed; most fail to gain wide adoption or reach sustainability [SE12]. In addition to these general difficulties, scientific software suffers from the fact that researchers who write and modify software often do not have formal training in software development. As a result, scientific software is often hampered by technical debt. These problems are mitigated by national initiatives to train scientists in software engineering skills, such as the UK's Software Sustainability Institute, as well as Software Carpentry [Wil14]. There is further work to be done in institutional design around filling this skills gap [KCN+16]. But it is known that computational thinking skills alone are not sufficient for successful

* *Corresponding author: spb413@nyu.edu*
§ *New York University School of Law*
‡ *Econ-ARK*

1. An example of an ABM is the Wolf Sheep Predation model, which is used to explore the stability of predator-prey ecosystems [Wil97].

scientific software. Software engineering skills are necessary to produce software that is usable beyond the lab or research group that originates it, which is a necessary path towards software sustainability [Ben19].

A second obstacle integrating software tools into scientific practice is that software-based learning requires additional education infrastructure. [SNLT18] document the challenges in providing JupyterHub with automatic grading extensions at universities and colleges; they find that many institutions do not have the resources or deep IT expertise necessary to build and maintain this infrastructure. The growing necessity of cloud-based computational notebooks for assignments and exploration in scientific education therefore raises concerns about social equity.

This paper explores these general themes through an analysis of Econ-ARK [CKK+18] as a case study. Econ-ARK is a domain specific software toolkit currently most widely used in Economics. Launched in 2014, the project has recently experienced a phase transition in development practices because of the onboarding of research software engineers. The collaborations between Economics professors and software engineers have revealed a broad scope of potential in computational research, publication, and pedagogy. It has also exposed how disciplinary training in Economics does not include many concepts necessary to realizing that potential. We conclude that the gaps between disciplinary training and the conditions for realizing this potential can be partially closed by framing software engineering as a research method.

Econ-ARK: Discipline Specifics

The Econ-ARK project [CKK+18] is a toolkit for the structural modeling of optimizing economic choices by heterogenous agents. A primary goal of its flagship software library HARK (Heterogenous Agent Research toolKit) is to support economic research into heterogeneous agent (HA) modeling [Hom06], which became a research priority after the 2008 financial crisis revealed the weaknesses in the then-dominant representative agent (RA) based paradigm.[2] It has been designed so that researchers and students can take a hands-on approach to economic modeling in software [CW18]. Econ-ARK is in some respects a port of Dynare [ABJ+11], an earlier computing library for economic models, into Python.

Econ-ARK lies roughly in the Papertian educational tradition, similar to other agent-based modeling software such as NetLogo [TW04] and Mesa [DMJK15]. However, in Econ-ARK models, agents optimize their behavior strategically with respect to predicted effects over time. In this respect, Econ-ARK has some characteristics of a reinforcement learning or artificial intelligence toolkit.

Example. A paradigmatic, simple example of the kind of problem studied using Econ-ARK is the microeconomic dynamic stochastic optimization problem of calculating the mathematically optimal amount to save [Car11].

This problem can be characterized by the equations:

$$\begin{aligned} U(c_t) &= \frac{c_t^{1-\rho}}{1-\rho} \\ m_{t+1} &= R(m_t - c_t) + p_{t+1} \\ p_{t+1} &= \gamma p_t \end{aligned}$$

where U is a utility function, ρ is a coefficient of risk aversion, c_t is the amount of resources the agent chooses to consume in each period t, m_t is the amount of market resources available to the agent at each time period, p_t is the level of income at each time period, γ is the growth rate of income over time, and R is a rate of return on savings.

These equations define a Markov Decision Problem (MDP), which can be transformed into a Bellman equation given a discount factor β:

$$V_t(m_t, p_t) = max_{c_t} U(c_t) + \beta V_{t+1}(m_{t+1}, p_{t+1})$$

The optimal consumer choice can be solved via dynamic programming.

However, it is possible to reduce the complexity of this problem significantly through mathematical analysis. Because income is growing geometrically, it is possible to remove one of the state variables p from the model, and solve for the MDP with the following transition function:

$$m_{t+1} = \frac{R}{\gamma}(m_t - \hat{c}_t) + 1$$

The consumption function $\hat{c}$ can then be solved in a reduced (1-dimensional) state space. The optimal consumption function for the original problem is then recoverable as $c_t = \hat{c}_t * p_t$. It is the goal of the Econ-ARK software to bundle the analytically reduced solution with the original model as a way of representing and making available the substantive knowledge gained in the mathematical derivation.

Models in HARK are, at a certain level of mathematical abstraction, equivalent to Markov Decision Problems (MDP). However, generic MDP software is not adequate for research in this field, for several reasons.

- **Substantive, policy-oriented structural modeling.** Unlike many recent fields of data science, in which generic model-fitting and machine-learning techniques are applied to a large data set for the purpose of maximizing predictive potential, this branch of Economics operates with relatively scarce data and a drive for model veracity. Besides the academic field of researchers, the intended audience for these models are national central banks and other policy-makers. For example, one policy application of these models is predicting the impact of the CARES stimulus bill on consumption [CCSW20]. These models are scientifically valued for their ability to approximate real social dynamics, and for their ability to build consensus towards policy-making, in addition to their goodness of fit to available data.
- **Analytical results informing solvers.** Like many other sciences, this branch of Economics has a theoretical component consisting in mathematical proofs about the models in question. In addition to providing interpretable insight into the invariant properties of a model, these results also inform the design of model solvers and the user experience. For example, a mathematical result might reveal under what parameter conditions a model has a degenerate solution; the software will warn the user if they attempt to solve the model in such a case. Elsewhere, an analytical result might provide a shortcut such that it is possible to write a solution algorithm with lower computational complexity than a generic one would have.
- **Continuous space decisions.** Most MDP solvers and simulators assume a discrete control and state space. The

2. These weaknesses had been known since the work of [Kir92].

economic problems studied using HARK are most often defined with continuous control and state spaces, and with continuous random variables as exogenous shocks. HARK therefore includes a variety of discretization and interpolation tools that support the transformation between discrete and continuous representations.

The upshot of these conditions is that Econ-ARK software is not only a tool for researchers doing empirical scientific work. Rather, its software is an encoding of substantive research results in mathematical theory. A software implementation, which integrates the results in a larger body of work and is subject to robust software testing, is an additional form of validation of the correctness and salience of a finding. This entails that the success of Econ-ARK will imply a practical change to the research field: students will study models that have been published in Python by researchers in order to learn insights about the economy.

Case Study: Roles in Econ-ARK

Econ-ARK has been broadly conceived as a collection of projects that supports this computational approach to education and research in economic structural modeling. The project has been organized around several different version-controlled software repositories. The software in these repositories is written mostly in Python, though there is also a great deal of expository content and sometimes older code in other languages such as MATLAB and Mathematica.

We have identified several different roles that people take on when interacting with Econ-ARK. The same individual or "natural person" might take on different roles at different times, but nevertheless these categories have been useful as ideal types [Hek83] with which to reason about requirements and skills.

Researcher. The role at the heart of the Econ-ARK system is that of the Researcher. This user is trying to advance the frontier of economic thinking by drawing on deep domain knowledge (Economics) as well as general training in computational and inferential thinking, applied math, and perhaps other fields. Research with Econ-ARK may be nebulously defined because while the question of how to implement a class of economic models efficiently and robustly in Python is a research question in its own right, these implementations are rarely considered first-order research contributions. Researchers work within a complex field of economic capital incentives (such as university salaries and grant funding) and symbolic capital incentives (scholar recognition for published work) [Bou04]. At the time of this article's publication, the institutional mechanisms for training and rewarding Economics researchers to work in the medium of robust software are few. As a consequence there is a skills gap: researchers often have programming ability, but not the software engineering and IT training that is necessary to fully realize the vision of the software's potential [CHH+13].

Publisher. One way to untie the Gordian knot of incentives around Econ-ARK research is to provide a more reliable and efficent path towards recognized scholarly publication that uses it. One proposal has been that economists begin a Journal of Open Source Economics [Isk19], modeled loosely on the Journal of Open Source Software (JOSS), which gives academic publication credit to the creators of scientific software tools. Preliminary efforts towards such a journal have been attempted through the Econ-ARK sub-project REMARK (Replications and Explorations Made using the ARK), which organizes contributed directories of material that meet a minimal 'publishable' standard of reproducibility. This approach has surfaced many challenges, mainly regarding the technical requirements of reliably hosting Python environments for each publishable unit, and managing dependencies across those environments. These technical challenges of *publication* require IT skills that are in general not available to researchers who may be technically capable of programming models that show substantive academic results.

Teacher and Student. In an academic context, the pedagogical use case is as important as the researcher's use case. While the researcher is building new models to communicate new discoveries, the teacher guides students to learn skills and ideas that are already known. Two of the hurdles faced by teachers attempting to use Econ-ARK pedagogically are the creation and grading of assignments and assisting students with the availability of an adequate computing environment that does not distract them from the course materials. Technical solutions have been developed for both hurdles. *nbgrader* enables the creation of assignments with Jupyter notebooks [Ham16] [BBB+19]. JupyterHub has been deployed to allow students to get around the hardware limitations of their laptops and the difficulties of setting up a local coding environment [Kim18]. Notably, both technical solutions, which have been developed only in the past few years, require skills that are not part of normal disciplinary training in economics. Economics professors currently require others to fill the social role that enables these tools to be useful.

Software engineer. The elephant in the room in all discussions of scientific software and computational education is that building and deploying robust software is its own complex field that often shares few disciplinary roots with the domain sciences. These skills are often specific to technologies that originated in industry or open source technology production, not in academia. For example, the version control system Git was not originally an academic project, but it nevertheless is now ubiquitously used for computational academic research through its popularization via GitHub. The workflow patterns of collaboratively developing software using GitHub and managing release cycles are not part of any conventional Economics curriculum, and yet researchers increasingly need to learn and use these in order to participate in computational research. Software engineering skills are not only useful for these infrastructural requirements of publication and pedagogy. Integrating new features, expressing substantive disciplinary material, and making these features available for new users requires these skills. In other words, software engineering skills are required to make a software project robust and reusable across many different labs and groups of researchers [Ben19]. This has led to calls in some places for a better supported and formalized role for Research Software Engineers [PHH16] [BHG+12].

This division of roles and skills raises some quandaries for computational economics. Publication, pedagogy, and the sustainability of the domain specific software library Econ-ARK all require software engineering skills. But there is no point at which new entrants into this discipline are trained in these skills. They must be learned informally by researchers who are not incentivized to do so, or they must be hired from an external talent pool trained in other disciplines or at another workplace.

This interrupts the cycle, from student to researcher to professor who teaches more students, which is necessary for the autonomy of Economics as a field of knowledge. If at every point in the process -- even at the point where new discoveries are

integrated into the core software library -- there is a dependence on an externally sourced skillset, then the discipline will fail to reproduce scholars with the competence to participate in its own field.

Case Study: Econ-ARK infrastructure

The Econ-ARK infrastructure is built around creating a sustainable community with respect to various use cases and the challenges of creating sustainable scientific software in Economics. We have discussed some of the challenges of bridging work across user roles of Researchers, Publishers, Professors and Software Engineers. Here. we illustrate these general points with examples from our software and infrastructure practices.

Decoupling scientific content from code. A lot of scientific code is written as part of academic research projects where the incentives aren't closely aligned with those of creating scientific software. The recent case of UK COVID microsimulation code [MRC] brings out a stronger need of creating scientific software with the correct incentives. The decision to draw the line between a research artifact and a software is a hard decision which varies a lot between different scientific domains and requires a high level overlap of the researcher, publisher and software engineer roles.

When scientific code written by researchers is geared towards the publishable end result like a paper, it can lead to short-sighted design choices that in a broader software context are known as "technical debt" [KNO12]. An illustration is this example of a difference between a script and a modular function [Sci].

```
# a research project to calculate the moving
# averages of two stocks

import pandas as pd

data = pd.read_csv('stocks_data.csv')

x = data['APPL'].rolling(window=5).mean()
y = data['GOOG'].rolling(window=5).mean()

print(x, y)
```

Running this script prints out the moving average time series of the two stocks. We can also create a software package which achieves the similar thing in a more modular way.

```
# move_avg.py

import pandas as pd

def calculate_MA(data, stock, days):
  # Calculates the moving average for a stock
  return data[stock].rolling(window=days).mean()
```

We can achieve similar results using our new package *move_avg*, but this isn't restricted to our specific hard coded variables (number of days, stock, input data).

```
import pandas as pd
from move_avg import calculate_MA

data = pd.read_csv('stocks_data.csv')
print(calculate_MA(data, 'APPL', 5))
print(calculate_MA(data, 'GOOG', 5))
```

Initial decisions like hard coding variables in the code while creating the research artifact (which happens in a lot of academic research projects) lead away from creating a well defined reusable scientific software library. This seems trivial for people with a software engineering background but not necessarily for others. We know this is a hard problem to solve in domain specific scientific code where the boundaries between a research paper and code could be blurry. To tackle this is Econ-ARK, we extracted generalized code from research artifacts to create our software package HARK [CKK+18] and maintained the research artifacts which heavily rely on HARK as REMARKs (Replications and Explorations Made using the ARK).

This decoupling exercise also helps with the reproducibility of research projects as it gives other researchers the necessary tools to examine the research artifacts. The decoupling can also enable the use of empirical data and model fitting techniques, expanding the functional scope of the original script.

Reproducible builds of scientific content. The reproducibility crisis has been plaguing academic research for some time and the current ecosystem of software packaging and distribution certainly does not help it. To tackle this in Econ-ARK we have used containerization technologies like Docker. Tools like Repo2Docker [Jup] further help us with creating reproducible builds of scientific content. Creating and working with these tools still requires a basic background with software engineering, and end users like students and researchers in economics may not have the required background. We made tools to lower the barrier by using pre-built containers and one-click (or one-command) reproducible research artifacts [EA]. This effort has required a strong overlap between Researchers and Software Engineers in a project. Pushing for reproducibility in the community benefits students by lowering the barriers to access research and publishers/researchers by creating tools required to address the reproducibility crisis.

Interactive scientific publication. The publication of the Econ-ARK-based analysis of the consumption response to the CARES Act [CCSW20] was accompanied by an online Dashboard[3] that allows users to change parameters of the model and visualize their impact on policy outcomes. This Dashboard was deployed using Binder and developed by an Econ-ARK Research Software Engineer. This dashboard supports the constructionist learning of the substance of the model. Here, that paradigm is applied to convey knowledge not to students, but to public policy makers and other economists.

This new way of presenting economic models may be more digestible to a wider audience than a traditional research publication. However, researchers are not trained to create these Dashboards as they are trained to write research papers. This limits the scholarly impact of domain specific research software, as many computational models are not being presented in this rich interactive way.

Teaching resources. To keep the wheels turning in a research discipline we require effective pedagogical resources, especially in domains which are increasingly using scientific software to further research. After creating pedagogical content we are faced with the next hard challenge of creating an effective teaching infrastructure. The crème de la crème of the SciPy community has faced installation problems with software packages and it is not hard to create a monster out of your local environment. But luckily tools like MyBinder and JupyterHub have drastically reduced the work required to set up a stable environment required for teaching courses that depend heavily on scientific software. At Econ-ARK we have used MyBinder (publicly and privately

3. https://mybinder.org/v2/gh/econ-ark/Pandemic/master?urlpath=voila%2Frender%2FCode%2FPython%2Fdashboard.ipynb

hosted) extensively for teaching graduate economics courses and it has significantly reduced the overhead required for local setup, especially for students who are the primary users of a domain specific scientific software like HARK. We have also effectively used containerization for standardizing student assignments which streamlines the work for both students and teachers.

Discussion

Is research software engineering becoming a core skill for research that involves writing code? The skills for navigating many practical elements of software engineering are necessary for equipping a digital classroom, effectively publishing results, and contributing new features to scientific libraries. Yet they are currently considered a peripheral part of disciplinary education in Economics. Researchers and professors are not taught these skills as part of their training as students. This contributes to a systemic skills gap between the discipline and technology.

One potential solution to this problem would be to introduce more software engineering training into the core curriculum for graduate students. Some Economics departments already offer a course on Computational Methods, analogous to earlier courses on Mathematical Methods, Econometrics, or other methods. As the pragmatic needs of computational methods increasingly require such activities as setting up local development environments, preparing cloud computing infrastructure, and utilizing autodocumentation, version control and package management tools, these techniques could be included as part of a computational methods curriculum.

This is a departure from both the computational thinking [Win06] approach, which emphasizes abstract, conceptual skills explicitly in contrast to the mechanical skills of programming, let alone software engineering. It is also a departure from constructionist learning [Pap82], in that the method of learning is not childlike play but what is instead most often considered a form of laborious work. Rather, it is perhaps best conceived and taught in the paradigm of situated learning [LW91], or an apprenticeship based model. In this model, students engage in "legitimate peripheral participation" by working with tools under the mentorship of experts, gradually becoming more central in the community of practice. This model has been applied to both software engineering education and open source community participation [YK03].

Preparing scientists with more general software engineering skills would pave the way for more general acceptance of computational narrative [PG15] as a core method in scientific practice. In the social sciences especially, this would open research fields to wider ranges of discoveries through computational methods. [Eps06] has argued that computational modeling in social science is the natural successor to game theoretic and rational choice modeling, which has a long social scientific history, allowing a wider range of models with greater realism and theoretical insight. While [Hom06] and [Tes06] have shown the applicability of these methods to economics in particular, progress has been limited by the lack of research software engineering skills available in the field. To unlock the potential of computational science, research software engineering must become recognized as a research method.

Another incentive for making software engineering more central as a research method for scientific practice is that mature software products are a vector for technology transfer from academic labs to the market [DR04]. As national funding agencies anticipate a pivot towards bringing scientific results to market a top priority [Amb20] it raises questions about what research methods are most commercially relevant.

We are definitely not the first push for more training to scientific researchers about general software design and best practices (software versioning, continuous integration, testing). Organizations like Software Carpentry [Wil14] have been successful in this domain. Creating sustainable domain specific scientific software requires a systematic decoupling of reusable library code from research artifacts so users from different backgrounds can successfully work with the software. Researchers writing code with knowledge about software design will have more success in creating a sustainable community. Our contribution in this paper is to discuss how software design can be reconceived as a scientific method, as opposed to a peripheral skill.

References

[ABJ+11] Stéphane Adjemian, Houtan Bastani, Michel Juillard, Ferhat Mihoubi, George Perendia, Marco Ratto, and Sébastien Villemot. Dynare: Reference manual, version 4. 2011.

[AD17] Ani Adhikari and John DeNero. Computational and Inferential Thinking: The Foundations of Data Science, 2017. URL: https://www.inferentialthinking.com/.

[Amb20] Mitch Ambrose. Lawmakers propose dramatic expansion of nsf to boost us technology, May 2020. URL: https://www.aip.org/fyi/2020/lawmakers-propose-dramatic-expansion-nsf-boost-us-technology.

[Bar16] Lorena A Barba. Computational thinking: I do not think it means what you think it means, 2016. URL: https://lorenabarba.com/blog/computational-thinking-i-do-not-think-it-means-what-you-think-it-means/.

[BBB+19] Douglas S Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas L Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, et al. nbgrader: A tool for creating and grading assignments in the jupyter notebook. *The Journal of Open Source Education*, 2(11), 2019. doi:10.21105/jose.00032.

[Ben19] Sebastian Benthall. Software incubator workshop: A synthesis, Feb 2019. URL: http://urssi.us/blog/2019/02/25/software-incubator-workshop-a-synthesis/.

[BHG+12] Rob Baxter, N Chue Hong, Dirk Gorissen, James Hetherington, and Ilian Todorov. The research software engineer. In *Digital Research Conference, Oxford*, pages 1–3, 2012.

[Bou04] Pierre Bourdieu. *Science of science and reflexivity*. Polity, 2004.

[CAC+09] Peter JA Cock, Tiago Antao, Jeffrey T Chang, Brad A Chapman, Cymon J Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, et al. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009. doi:10.1093/bioinformatics/btp163.

[Car11] Christopher D Carroll. Solution methods for microeconomic dynamic stochastic optimization problems, 2011. URL: http://www.econ.jhu.edu/people/ccarroll/solvingmicrodsops.pdf.

[CCSW20] Christopher D Carroll, Edmund Crawley, Jiri Slacalek, and Matthew N White. Modeling the consumption response to the cares act. *COVID Economics*, 2020.

[CHH+13] Stephen Crouch, Neil Chue Hong, Simon Hettrick, Mike Jackson, Aleksandra Pawlik, Shoaib Sufi, Les Carr, David De Roure, Carole Goble, and Mark Parsons. The software sustainability institute: changing research software attitudes and practices. *Computing in Science & Engineering*, 15(6):74–80, 2013. doi:10.1109/MCSE.2013.133.

[CKK+18] Christopher D. Carroll, Alexander M. Kaufman, Jacqueline L. Kazil, Nathan M. Palmer, and Matthew N. White. The Econ-ARK and HARK: Open Source Tools for Computational Economics. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 25 – 30, 2018. doi:10.25080/Majora-4af1f417-004.

[CW18] Christopher D Carroll and Matthew N White. Hands-on heterogeneous agent macroeconomics, 2018. URL: https://safe-frankfurt.de/fileadmin/user_upload/editor_common/Events/Chris_Carrol_Syllabus.pdf.

[DMJK15] David Masad and Jacqueline Kazil. Mesa: An Agent-Based Modeling Framework. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 51 – 58, 2015. doi:10.25080/Majora-7b98e3ed-009.

[DR04] Jean-Michel Dalle and Guillaume Rousseau. Toward collaborative open-source technology transfer. *Collaboration, Conflict and Control*, pages 34–42, 2004. doi:10.1049/ic:20040262.

[EA] Econ-ARK. BufferStockTheory reproduce. URL: https://github.com/llorracc/BufferStockTheory/blob/master/reproduce.sh.

[Eps06] Joshua M Epstein. *Generative social science: Studies in agent-based computational modeling*. Princeton University Press, 2006. doi:10.23943/princeton/9780691158884.001.0001.

[EVDSLB19] Eric Van Dusen, Anthony Suen, Alan Liang, and Amal Bhatnagar. Accelerating the Advancement of Data Science Education. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 1 – 4, 2019. doi:10.25080/Majora-7ddc1dd1-000.

[Guz08] Mark Guzdial. Education paving the way for computational thinking. *Communications of the ACM*, 51(8):25–27, 2008. doi:10.1145/1378704.1378731.

[Ham16] Jessica B Hamrick. Creating and grading ipython/jupyter notebook assignments with nbgrader. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 242–242, 2016. doi:10.1145/2839509.2850507.

[Hek83] Susan J Hekman. Weber's ideal type: A contemporary reassessment. *Polity*, 16(1):119–137, 1983. doi:10.2307/3234525.

[Hom06] Cars H Hommes. Heterogeneous agent models in economics and finance. *Handbook of Computational Economics*, 2:1109–1186, 2006. doi:10.1016/s1574-0021(05)02023-x.

[Isk19] Fedor Iskhakov. The journal of open source economics journal charter, 2019. URL: https://github.com/joseconomics/JOSEcon-Project-Charter/blob/master/josecon_charter.pdf.

[Jor16] Michael I Jordan. On computational thinking, inferential thinking and data science. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 47–47, 2016. doi:10.1145/2935764.2935826.

[Jup] Jupyter. repo2docker. URL: https://github.com/jupyter/repo2docker.

[KCN+16] Daniel Katz, Sou-Cheng Choi, Kyle Niemeyer, James Hetherington, Frank Löffler, Dan Gunter, Ray Idaszak, Steven Brandt, Mark Miller, Sandra Gessing, et al. Report on the third workshop on sustainable software for science: Practice and experiences (wssspe3). *Journal of Open Research Software*, 4(1), 2016. doi:10.5334/jors.118.

[Kim18] Alicia Kim. The jupyterhub journey: Starting small and scaling up, May 2018. URL: https://data.berkeley.edu/news/jupyterhub-journey-starting-small-and-scaling.

[Kir92] Alan P Kirman. Whom or what does the representative individual represent? *Journal of economic perspectives*, 6(2):117–136, 1992.

[KNO12] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012. doi:10.1109/ms.2012.167.

[LW91] Jean Lave and Etienne Wenger. *Situated learning: Legitimate peripheral participation*. Cambridge University Press, 1991. doi:10.1017/cbo9780511815355.

[McK11] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[MPSC+13] Stuart Mumford, David Pérez-Suárez, Steven Christe, Florian Mayer, and Russell J. Hewett. SunPy: Python for Solar Physicists. In Stéfan van der Walt, Jarrod Millman, and Katy Huff, editors, *Proceedings of the 12th Python in Science Conference*, pages 70 – 73, 2013. doi:10.25080/Majora-8b375195-00c.

[MRC] UK MRC. Covidsim microsimulation model. URL: https://github.com/mrc-ide/covid-sim.

[Pap82] Seymour Papert. Mindstorms. *NY: Basic Books*, 1982. doi:10.1007/978-3-0348-5357-6.

[Pei07] Jonathan W Peirce. Psychopy—psychophysics software in python. *Journal of Neuroscience Methods*, 162(1-2):8–13, 2007. doi:10.1016/j.jneumeth.2006.11.017.

[PG15] Fernando Perez and Brian E Granger. Project jupyter: Computational narratives as the engine of collaborative data science, 2015. URL: http://archive.ipython.org/JupyterGrantNarrative-2015.pdf.

[PHH16] Olivier Philippe, Neil Chue Hong, and Simon Hettrick. Preliminary analysis of a survey of uk research software engineers. In *4th Workshop on Sustainable Software for Science: Practice and Experience*, 2016.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[ROP90] Mitchel Resnick, Stephen Ocko, and Seymour Papert. *LEGO/logo–learning through and about design*. Epistemology and Learning Group, MIT Media Laboratory Cambridge, MA, 1990.

[RTG+13] Thomas P Robitaille, Erik J Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis, Adam Ginsburg, Adrian M Price-Whelan, Wolfgang E Kerzendorf, et al. Astropy: A community python package for astronomy. *Astronomy & Astrophysics*, 558:A33, 2013.

[Sci] SciPy. Scipy lecture notes. URL: https://scipy-lectures.org/intro/language/reusing_code.html.

[SE12] Charles M Schweik and Robert C English. *Internet success: a study of open-source software commons*. MIT Press, 2012. doi:10.7551/mitpress/9780262017251.001.0001.

[SGB13] Amber Settle, Debra S Goldberg, and Valerie Barr. Beyond computer science: computational thinking across disciplines. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, pages 311–312, 2013. doi:10.1145/2462476.2462511.

[SNLT18] Anthony Suen, Laura Norén, Alan Liang, and Andrea Tu. Equity, Scalability, and Sustainability of Data Science Infrastructure. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 15 – 17, 2018. doi:10.25080/Majora-4af1f417-002.

[Tes06] Leigh Tesfatsion. Agent-based computational economics: A constructive approach to economic theory. *Handbook of Computational Economics*, 2:831–880, 2006. doi:10.1016/s1574-0021(05)02016-2.

[TW04] Seth Tisue and Uri Wilensky. Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, volume 2004, pages 7–9, 2004.

[WCV11] Stéfan 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.

[Wil97] Uri Wilensky. Netlogo wolf sheep predation model, 1997. URL: http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation.

[Wil14] Greg Wilson. Software carpentry: lessons learned. *F1000Research*, 3, 2014. doi:10.12688/f1000research.3-62.v2.

[Win06] Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006. doi:10.1145/1118178.1118215.

[YK03] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 25th International Conference on Software Engineering, 2003*, pages 419–429. IEEE, 2003. doi:10.1109/icse.2003.1201220.

Falsify your Software: validating scientific code with property-based testing

Zac Hatfield-Dodds[‡*]

Abstract—Where traditional example-based tests check software using manually-specified input-output pairs, property-based tests exploit a general description of valid inputs and program behaviour to automatically search for falsifying examples. Given that Python has excellent property-based testing tools, such tests are often *easier* to work with and routinely find serious bugs that all other techniques have missed.

I present four categories of properties relevant to most scientific projects, demonstrate how each found real bugs in Numpy and Astropy, and propose that property-based testing should be adopted more widely across the SciPy ecosystem.

Index Terms—methods, software, validation, property-based testing

Introduction

Much research now depends on software for data collection, analysis, and reporting; including on software produced and maintained by researchers. This has empowered us enormously: it is hard to imagine an analysis that was possible *at all* a generation ago which could not be accomplished quickly by a graduate student today.

Unfortunately, this revolution in the power and sophistication of our software has largely outstripped work on validation. While it would be unthinkable to publish research based on custom-built and unvalidated physical instruments, this is routine in software. As an effect, [Soe15] estimates that

> Any reported scientific result could very well be wrong if data have passed through a computer, and these errors may remain largely undetected[1]. It is therefore necessary to greatly expand our efforts to validate scientific software.

I argue that property-based testing [Mac] is more effective for validation of Python programs than using only traditional example-based tests[2], and support this argument with a variety of examples from well-known scientific Python projects.

This is a recent development: while the concept of property-based testing dates back to 1999 [CH00], early tools required deep computer-science expertise. Since 2015, Hypothesis [MHDC19] has made state-of-the-art testing technology available *and accessible to non-experts* in Python, and has added multiple features designed for testing scientific programs[3] since 2019.

What is property-based testing?

Where example-based tests check for an exact expected output, property-based tests make *less precise* but *more general* assertions. By giving up hand-specification of the expected output, we gain tests that can be run on a wide range of inputs.

This generality also guides our tests to the right level of abstraction, and gives clear design feedback: where example-based tests map one input to one output regardless of complexity, every special case or interacting feature has to be addressed. Clean abstractions which allow you to say "for all ...", without caveats, stand in clear contrast and are a pleasure to test.

Tests which use random data are usually property-based, but using a library designed for the task has several advantages:

- a concise and expressive interface for describing inputs
- tests are never flaky - failing examples are cached and replayed, even if the test failed on a remote build server
- automatic shrinking facilitates debugging by presenting a minimal failing example for each distinct error

Automating these tedious tasks makes finding bugs considerably faster, and debugging both easier and more fun. When Hypothesis world-class shrinker [MD] hands you a failing example, you *know* that every feature is relevant - if any integer could be smaller (without the test passing) it would be; if any list could be shorter or sorted (ditto) it would be, and so on.

Why is this so effective?

Examples I wouldn't think of reveal bugs I didn't know were possible. It turns out that general descriptions of your data have several advantages over writing out specific examples, and that these are even stronger for research code.

Input descriptions are concise. Writing out a Hypothesis "strategy"[4] describing objects like a Numpy array or a Pandas dataframe is often less code than a single instance, and clearly expresses to readers what actually matters. The sheer tedium of writing out representative test cases, without resorting to literally random data, is a substantial deterrent to testing data analysis code; with property-based testing you don't have to.

* Corresponding author: zac.hatfield.dodds@anu.edu.au
‡ Australian National University

1. and indeed [BNNL+19] reported such a bug, affecting around 160 papers.
2. i.e. common workflows using `pytest`. If you have no automated tests at all, fix that first, but your second test could reasonably use Hypothesis.
3. e.g. numeric-aware error reporting, first-class support for array shapes including broadcasting and gufunc signatures, dtypes and indexers, etc.

The system is designed to find bugs. Hypothesis also comes packed with well-tuned heuristics and tools for finding bugs which uniformly random data would almost never find - literal 'edge cases' in the space of possible inputs. In one memorable case a new user was testing coordinate transformation logic, which fails for singular matrices (a set of measure zero). Hypothesis knows nothing at all about matrices, or even the topic of the test, but promptly generated a failing example anyway.

Describe once, test everywhere. In a codebase with M variations on the core data structures and N features, example-based tests have $M \times N$ tests to write - and it's all too easy to forget to test the interaction of lesser-known features[5]. With property-based tests those M variations can be designed into the strategies which describe your data, scaling as $M + N$ and ensuring that nothing is forgotten[6].

This scaling effect makes effective testing much easier for new contributors, who do not need to consider all possible feature interactions - they will arise naturally from the shared input descriptions. For example, Hypothesis' Numpy extension includes tools to describe arrays, array shapes including broadcasting and generalised ufunc signatures, scalar and structured dtypes, and both basic and advanced indexing. Thinking carefully about what inputs *should* be supported is usually a valuable exercise in itself!

We face multiple sources of uncertainty. When experimental results come out weird, unpicking the unexpected behaviour of your research domain from the possibility of equipment error or software bugs is hard enough already. Property-based tests let you verify more general behaviours of your code, and focus on the domain rather than implementation details.

Properties and Case Studies

In this section I present four categories of properties. While not an exhaustive list[7], they are relevant to a very wide range of scientific software - and when tested often uncover serious errors.

I also present case studies of real-world bugs[8] from the SciPy stack, especially from foundational libraries like Numpy [Oli06] and Astropy [ART+13] [PWSG+18]. While seriously under-resourced given their importance to essentially all research in their fields [Num] [ea16], they are well-engineered and no more defect-prone than any comparable software. *If it can happen to them, it can certainly happen to you.*

The bugs presented below were each discovered, reported, and fixed within a few days thanks to a community-driven and open source development model; and projects from Astropy to Xarray - via Numpy and Pandas - have begun to adopt property-based tests.

Outputs within expected bounds

For many functions, the simplest property to check is that their output is within some expected bound. These may be computational or logical bounds like the limits of probability as $[0, 1]$, or might be physical bounds like the temperature $-273.15°C$.

Consider the `softmax` function, as described by the SciPy documentation[9]. This function is often used to convert a vector of real numbers into a probability distribution, so we know that sum should always be (approximately) equal to one. Let's test that with an example-based and a property-based test:

```
from hypothesis import given, strategies as st
import hypothesis.extra.numpy as npst

def softmax(x):
    return np.exp(x) / np.exp(x).sum()

def test_softmax_example():
    assert softmax(np.arange(5)).sum() == 1

@given(npst.arrays(
    dtype=float,
    shape=npst.array_shapes(),
    elements=st.floats(
        allow_nan=False, allow_infinity=False
    )
))
def test_softmax_property(arr):
    total = softmax(arr).sum()
    np.testing.assert_almost_equal(total, 1)
```

While our example-based test passes for small arrays of small integers, the naive algorithm is numerically unstable! Our property-based test fails almost instantly, showing us the minimal example of overflow with `np.exp([710.])`. If we instead use `np.exp(x - x.max())`, the test passes.

I will not argue that this kind of testing can substitute for numerical analysis, but rather that it can easily be applied to routines which would otherwise not be analysed at all.

A more sophisticated example of bounds testing comes from recent work in Astropy[10], using Hypothesis to check that conversions between different time scales did not unexpectedly lose precision[11]. Astropy contributors wrote custom strategies to incorporate bias towards leap-seconds (unrepresentable in `datetime.datetime`), and an `assert_almost_equal` helper which uses `hypothesis.target()` to guide the search process towards larger errors.

These tests found that round-trip conversions could be off by up to twenty microseconds over several centuries[12] due to loss of precision in `datetime.timedelta.total_seconds()`. This effort also contributed to improved error reporting around the 'threshold problem'[13], where a minimal failing example does not distinguish between subtle and very serious bugs.

Round-trip properties

Whenever you have a pair of inverse functions, think of round-trip testing. If you save and then load data, did you lose any? If

4. for historical reasons, Hypothesis calls input descriptions 'strategies'

5. e.g. signalling NaNs, zero-dimensional arrays, structured Numpy dtypes with field titles in addition to names, explicit dtype padding or endianness, etc. Possible *combinations* of such features are particularly neglected.

6. test 'fixture' systems scale similarly, but are less adaptable to individual tests and can only be as effective as the explicit list of inputs they are given.

7. a notable omission is the 'null property', where you execute code on valid inputs but do not make any assertions on its behaviour. This is shockingly effective at triggering internal errors, even before use of assertions in the code under test - and a simple enough technique to explain in a footnote!

8. preferring those which can be demonstrated and explained in only a few lines, though we have found plenty more which cannot.

9. docs.scipy.org/doc/scipy/reference/generated/scipy.special.softmax.html

10. culminating in github.com/astropy/astropy/pull/10373

11. as background, Python's builtin `datetime.datetime` type represents time as a tuple of integers for year, month, ..., seconds, microseconds; and assumes UTC and the current Gregorian calendar extended in both directions. By contrast `astropy.time.Time` represents time with a pair of 64-bit floats; supports a variety of civil, geocentric, and barycentric time scales; and maintains sub-nanosecond precision over the age of the universe!

12. while a 20us error might not sound like much, it is a *hundred billion times* the quoted precision, and intolerable for e.g. multi-decade pulsar studies.

13. described in hypothesis.works/articles/threshold-problem/ and addressed by github.com/HypothesisWorks/hypothesis/pull/2393

you convert between two formats, or coordinate systems, can you convert back?

These properties are remarkably easy to test, vitally important, and often catch subtle bugs due to the complex systems interactions. It is often worth investing considerable effort to describe *all* valid data, so that examples can be generated with very rare feature combinations.

If you write only one test based on this paper, *try to save and load any valid data*.

I have consistently found testing IO round-trips to be among the easiest and most rewarding property tests I write. My own earliest use of Hypothesis came after almost a month trying to track down data corruption issues in multi-gigabyte PLY files. Within a few hours I wrote a strategy to generate PLY objects, executed the test, and discovered that our problems were due to mishandling of whitespace in the file header[14].

Even simple tests are highly effective though - consider as an example

```
@given(st.text(st.characters())
       .map(lambda s: s.rstrip("\x00")))
def test_unicode_arrays_property(string):
    assert string == np.array([string])[0]
```

This is a more useful test that it might seem: after working around null-termination of strings, we can still detect a variety of issues with length-aware dtypes, Unicode version mismatches, or string encodings. A very similar test did in fact find an encoding error[15], which was traced back to a deprecated - and promptly removed - compatibility workaround to support 'narrow builds' of Python 2.

Differential testing

Running the same input through your code and through a trusted - or simply different - implementation is another widely applicable property: any difference in the outputs indicates that *at least* one of them has a bug. Common sources of alternative implementations include:

Another project or language. If you aim to duplicate functionality from an existing project, you can check that your results are identical for whatever overlap exists in the features of the two projects. This might involve cross-language comparisons, or be as simple as installing an old version of your code from before a significant re-write.

A toy or brute-force implementation which only works for small inputs might be out of the question for 'production' use, but can nonetheless be useful for testing. Alternatively, differential testing can support ambitious refactoring or performance optimisations - taking existing code with "obviously no bugs" and using it to check a faster version with "no obvious bugs".

Varying unrelated parameters such as performance hints which are not expected to affect the calculated result. Combining this and the previous tactic, try comparing single-threaded vs. multi-threaded mode - while some care is required to ensure determinism it is often worth the effort.

As our demonstration, consider the `numpy.einsum` function and two tests. The example-based test comes from the Numpy test suite; and the property-based test is a close translation - it still requires two-dimensional arrays, but allows the shapes and contents to vary. Note that both are differential tests!

```
def test_einsum_example():
    p = np.ones(shape=(10, 2))
    q = np.ones(shape=(1, 2))
    assert_array_equal(
        np.einsum("ij,ij->j", p, q, optimize=True),
        np.einsum("ij,ij->j", p, q, optimize=False)
    )

@given(
    data=st.data(),
    dtype=npst.integer_dtypes(),
    shape=npst.array_shapes(min_dims=2, max_dims=2),
)
def test_einsum_property(data, dtype, shape):
    p = data.draw(npst.arrays(dtype, shape))
    q = data.draw(npst.arrays(dtype, shape))
    assert_array_equal( ... )  # as above
```

When an optimisation to avoid dispatching to `numpy.tensordot` over a dimension of size one was added, the example-based test kept passing - despite the bug `if 1 in operands[n]` instead of `if 1 in operands[n].shape`[16]. This bug could only be triggered with `optimize=True` and an input array with a dimension of size one, *xor* containing the integer `1`. This kind of interaction is where property-based testing really shines.

There's another twist to this story though: the bug was actually identified downstream of Numpy, when Ryan Soklaski was testing that `Tensors` from his auto-differentiation library `MyGrad` [Sok] were in fact substitutable for Numpy arrays[17]. He later said of property-based tests that[18]

> It would have been impossible for me to implement a trustworthy autograd library for my students to learn from and contribute to if it weren't for Hypothesis.

Metamorphic properties

A serious challenge when testing research code is that the correct result may be genuinely unknown - and running the shiny new simulation or analysis code is the only way to get any result at all. One very powerful solution is to compare several input-output pairs, instead of attempting to analyse one in isolation:

> A test oracle determines whether a test execution reveals a fault, often by comparing the observed program output to the expected output. This is not always practical... Metamorphic testing provides an alternative, where correctness is not determined by checking an individual concrete output, but by applying a transformation to a test input and observing how the program output "morphs" into a different one as a result. [SFSR16]

Let's return to `softmax` as an example. We can state general properties about a single input-output pair such as "all elements of the output are between zero and one", or "the sum of output elements is approximately equal to one"[19]. A metamorphic property we could test is scale-invariance: multiplying the input elements by a constant factor should leave the output approximately unchanged.

```
@given(arr=..., factor=st.floats(-1000, 1000))
def test_softmax_metamorphic_property(arr, factor):
```

14. github.com/dranjan/python-plyfile/issues/9
15. github.com/numpy/numpy/issues/15363
16. github.com/numpy/numpy/issues/10930
17. making `test_einsum_property` a differential test derived from a derivative auto-differentiator.
18. github.com/HypothesisWorks/hypothesis/issues/1641

```
result = softmax(arr)
scaled = softmax(arr * factor)
np.testing.assert_almost_equal(result, scaled)
```

Astropy's tests for time precision include metamorphic as well as round-trip properties: several assert that given a `Time`, adding a tiny timedelta then converting it to another time scale is almost equal to converting and then adding.

Metamorphic properties based on domain knowledge are particularly good for testing "untestable" code. In bioinformatics, [CHLX09] presents testable properties for gene regulatory networks and short sequence mapping[20], and found a bug attributable to the *specification* - not just implementation errors. METTLE [XZC+20] proposes eleven generic metamorphic properties for unsupervised machine-learning sytems[21], and studies their use as an aid to end-users selecting an appropriate algorithm in domains from LIDAR to DNA sequencing.

Conclusion

Example-based tests provide anecdotal evidence for validity, in that the software behaves as expected on a few known and typically simple inputs. Property-based tests require a precise description of possible inputs and a more general specification, but then automate the search for falsifying counter-examples. They are quick to write, convenient to work with, and routinely find serious bugs that all other techniques had missed.

I argue that this Popperian approach is superior to the status quo of using only example-based tests, and hope that the property-based revolution comes quickly.

Acknowledgements

Thanks to David MacIver and the many others who have contributed to Hypothesis; to Hillel Wayne, Kathy Reid, and Ryan Soklaski for their comments on an early draft of this paper; to Anne Archibald for her work with threshold tests; and to the many mantainers of the wider Python ecosystem.

References

[ART+13] Astropy Collaboration, T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf, A. Conley, N. Crighton, K. Barbary, D. Muna, H. Ferguson, F. Grollier, M. M. Parikh, P. H. Nair, H. M. Unther, C. Deil, J. Woillez, S. Conseil, R. Kramer, J. E. H. Turner, L. Singer, R. Fox, B. A. Weaver, V. Zabalza, Z. I. Edwards, K. Azalee Bostroem, D. J. Burke, A. R. Casey, S. M. Crawford, N. Dencheva, J. Ely, T. Jenness, K. Labrie, P. L. Lim, F. Pierfederici, A. Pontzen, A. Ptak, B. Refsdal, M. Servillat, and O. Streicher. Astropy: A community Python package for astronomy. *Astronomy & Astrophysics*, 558, October 2013. doi:10.1051/0004-6361/201322068.

[BNNL+19] Jayanti Bhandari Neupane, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Philip G. Williams. Characterization of leptazolines a–d, polar oxazolines from the cyanobacterium leptolyngbya sp., reveals a glitch with the "willoughby–hoye" scripts for calculating nmr chemical shifts. *Organic Letters*, 21(20):8449–8453, 2019. URL: https://doi.org/10.1021/acs.orglett.9b03216, doi:10.1021/acs.orglett.9b03216.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000. URL: https://doi.org/10.1145/357766.351266, doi:10.1145/357766.351266.

[CHLX09] Tsong Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):24, 2009. URL: https://doi.org/10.1186/1471-2105-10-24, doi:10.1186/1471-2105-10-24.

[ea16] Demitri Muna et al. The astropy problem, 2016. arXiv:1610.03159.

[LSH+] Benjamin Lee, Reva Shenwai, Zongyi Ha, Michael B. Hall, and Vaastav Anand. Hypothesis-Bio. URL: https://github.com/Lab41/hypothesis-bio.

[Mac] David MacIver. In Praise of Property-Based Testing. URL: https://increment.com/testing/in-praise-of-property-based-testing/.

[MD] David MacIver and Alastair Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. to be published in https://2020.ecoop.org/details/ecoop-2020-papers/13/Test-Case-Reduction-via-Test-Case-Generation-Insights-From-the-Hypothesis-Reducer. URL: https://drmaciver.github.io/papers/reduction-via-generation-preview.pdf.

[MHDC19] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019. URL: https://doi.org/10.21105/joss.01891, doi:10.21105/joss.01891.

[Num] NumFocus. Why is numpy only now getting funded? URL: https://numfocus.org/blog/why-is-numpy-only-now-getting-funded.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PWSG+18] A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, and et al. The astropy project: Building an open-science project and status of the v2.0 core package. *The Astronomical Journal*, 156(3):123, Aug 2018. URL: http://dx.doi.org/10.3847/1538-3881/aabc4f, doi:10.3847/1538-3881/aabc4f.

[SFSR16] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[Soe15] David A. W. Soergel. Rampant software errors may undermine scientific results. *F1000Research*, 3:303, July 2015. URL: https://doi.org/10.12688/f1000research.5930.2, doi:10.12688/f1000research.5930.2.

[Sok] Ryan Soklaski. Mygrad. URL: https://github.com/rsokl/MyGrad.

[XZC+20] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. Mettle: A metamorphic testing approach to assessing and validating unsupervised machine learning systems. *IEEE Transactions on Reliability*, page 1–30, 2020. URL: http://dx.doi.org/10.1109/TR.2020.2972266, doi:10.1109/tr.2020.2972266.

19. or even `np.argsort(arr) == np.argsort(softmax(arr))`

20. e.g. K-means clustering

21. for which `hypothesis-bio` [LSH+] provides many useful data-generation strategies

Towards an Unsupervised Spatiotemporal Representation of Cilia Video Using A Modular Generative Pipeline

Meekail Zain[‡§†], Sonia Rao[‡†], Nathan Safir[‡], Quinn Wyner[§], Isabella Humphrey[‡§], Alex Eldridge[§], Chenxiao Li[‖], BahaaEddin AlAila[‡**], Shannon Quinn[‡¶*]

Abstract—Motile cilia are a highly conserved organelle found on the exterior of many human cells. Cilia beat in rhythmic patterns to transport substances or generate signaling gradients. Disruption of these patterns is often indicative of diseases known as ciliopathies, whose consequences can include dysfunction of macroscopic structures within the lungs, kidneys, brain, and other organs. Characterizing ciliary motion phenotypes as healthy or diseased is an essential step towards diagnosing and differentiating ciliopathies. We propose a modular generative pipeline for the analysis of cilia video data so that expert labor may be supplemented for this task. Our proposed model is divided into three modules: preprocessing, appearance, and dynamics. The preprocessing module augments the initial data, and its output is fed frame-by-frame into the generative appearance model which learns a compressed latent representation of the cilia. The frames are then embedded into the latent space as a low-dimensional path. This path is fed into the generative dynamics module, which focuses only on the motion of the cilia. Since both the appearance and dynamics modules are generative, the pipeline itself serves as an end-to-end generative model. This thorough and versatile model allows experts to spend less time caught in the minutiae of cilia biopsy analysis, while also enabling new insights by quantifying subtle patterns that would be otherwise difficult to categorize.

Index Terms—Machine Learning, Data Science, Video Analysis, Generative Modeling, Variational Autoencoder, Modular, Pipeline

Introduction

Motile cilia are organelles commonly found throughout the human body, such as in the bronchial and nasal passages [HGGRD99][SS90]. Cilia beat in synchronous, rhythmic patterns to expel foreign matter, collectively forming the mucociliary defense, a vital mechanism for sinopulmonary health [BMO17]. Ciliopathies are genetic disorders which can adversely affect the motion of cilia [FL12]. Disorders resulting from the disruption of ciliary motion range from sinopulmonary diseases such as primary ciliary dyskinesia (PCD) [OCH+07] to mirror symmetric organ placement and situs inversus [CA17] or randomized left-right organ placement as in heterotaxy [GZT+14]. Precise diagnosis of patients exhibiting abnormal ciliary motion prior to surgery may provide clinicians with opportunities to institute prophylactic respiratory therapies to prevent complications. Therefore, the study of ciliary motion may have a broad clinical impact.

Visual examination of the ciliary waveform by medical professionals is critical in diagnosing ciliary motion defects, but such manual analysis is highly subjective and prone to error [RWH+14] [KSC17]. This approach also precludes the possibility of cross-institutional and longitudinal studies which include assessment of ciliary motion. Therefore, we aim to develop an unsupervised, computational approach to analyze ciliary motion, developing a quantitative "library" of well-defined, clinically relevant ciliary motion phenotypes. Clustering and classification are established problems in machine learning. However, their applications to ciliary waveform analysis are difficult, as cilia exhibit subtle, rotational, non-linear motion [QFLC11]. While attempts have been made at addressing this problem, we note that generic dynamics models fail to classify and cluster this type of motion accurately or meaningfully, and are insufficient for generating a semantically potent representation. We thus apply a novel machine learning approach to create an underlying representation which then can be used for downstream tasks such as classification and clustering, and any other tasks that experts may deem necessary. Furthermore, we avoid using labeled data—specifically videos anotated based on the health/type of ciliary motion displayed—in order to free the model from systematic assumptions naturally imposed by labels: the choice of labels themselves can inadvertently limit the model by asserting that all data *must* conform to those exact labels. An unsupervised model has the freedom to discover potential semantically meaningful patterns and phenotypes that fall outside current clinical thinking. Furthermore, an unsupervised model is independent of expert input. Pragmatically, an unsupervised model can be trained and used directly after data acquisition, rather than having to wait on expert labeling. This simultaneously reduces the barriers to access as a scientific tool, and the associated expenses of use.

Our approach is to create a pipeline that learns a low-dimensional representation of ciliary motion on unlabeled data. The model we propose considers the spatial and temporal dimensions of ciliary motion separately. The pipeline encodes each

† *These authors contributed equally.*
‡ *Computer Science Department, Franklin College of Arts and Sciences*
§ *Mathematics Department, Franklin College of Arts and Sciences*
‖ *Comparative Biomedical Sciences, College of Veterinary Medicine*
** *Institute for Artificial Intelligence, Franklin College of Arts and Sciences*
* *Corresponding author: spq@uga.edu*
¶ *Cellular Biology Department, Franklin College*

frame of the input video and then encodes the paths between frames in the latent space. The low-dimensional latent space in this pipeline will have semantic significance, and thus the distribution and clustering of points in the latent space should be meaningful for those studying ciliary motion and its connection to ciliopathies.

Related Works

A computational method for identifying abnormal ciliary motion patterns was proposed by Quinn 2015 [QZD+15]. The authors hypothesize ciliary motion as an instance of a dynamic texture, which are rhythmic motions of particles subjected to stochastic noise [DCWS03] and include familiar patterns such as flickering flames, rippling water, and grass in the wind. Each instance of dynamic texture contains a small amount of stochastic behavior altering an otherwise consistent visual pattern. The authors chose to consider ciliary motion as a dynamic texture as it consists of rhythmic behavior subject to stochastic noise that collectively determine the beat pattern. They then used autoregressive (AR) representations of optical flow features that were fed into a support vector machine classifier to decompose high-speed digital videos of ciliary motion into "elemental components," or quantitative descriptors of the ciliary motion, and classify them as normal or abnormal.

While this study proved there is merit in treating ciliary motion as a dynamic texture, the use of an AR model for the classification task imposed some critical limitations. While AR models are often used in representing dynamic textures, they are primarily used in distinguishing distinct dynamic textures (e.g., rippling water from billowing smoke), rather than identifying different instances of the same texture (e.g., cilia beating normally versus abnormally). Additionally, AR models impose strong parametric assumptions on the underlying structure of the data, rendering AR models incapable of capturing nonlinear interactions. Lastly, even though the majority of the pipeline is automated, their study relied on clinical experts to manually annotate the video data with regions of interest (ROIs) in order to serve as ground truth for the inference. Drawing ROIs required specialized labor, increasing the cost and time of clinical operations. This is also potentially problematic in that expert drawn ROIs introduce the same subjective bias that the study is ostensibly attempting to remove.

The model proposed by Quinn 2015 was improved upon by Lu 2018 [LMZ+18], the latter attempt using stacked Fully Convolutional DenseNets [HLW16] and Long Short-Term Memory (LSTM) networks [GSC99]. Densely Connected Convolutional Networks, referred to as DenseNets, do not make strong parametric or linear assumptions about the underlying data, allowing more complex behavior to be captured. Once Lu 2018 extract segmentation masks using their 74-layer FCDenseNet, ciliary motion is treated as a time series using convolutional long short-term memory (Conv-LSTM) networks, a specific type of recurrent neural network (RNN), to model the long-term temporal dependencies in the data.

We aim to build upon these studies by developing a fully unsupervised approach to characterizing ciliary motion phenotypes. This pipeline is advantageous in that it does not need hand-drawn ROI maps nor a labeled dataset as training data. While clinicians acknowledge the existence of distinct ciliary waveform phenotypes beyond "normal" and "abnormal", experts lack standard guidelines for qualitatively or quantitatively categorizing ciliary beat pattern. Additionally, experts may not observe the level of quantitative detail required to associate complex motion phenotypes with specific ciliopathies and genetic mutations [QZD+15]. Thus, we shift away from a classification-style task (classifying abnormal versus normal ciliary motion) to a representational learning task to generate meaningful, low-dimensional representations of ciliary motion. Unsupervised representation learning enables a model to learn families of complex ciliary motion phenotypes beyond the normal-abnormal binary.

Methods

Our proposed model is divided into three modules: preprocessing, appearance, and dynamics. The preprocessing module primarily serves to supplement input data by generating segmentation masks and extracting dense optical flow vector fields and pertinent differential quantities. Segmentation masks are used to limit spatial representation learning to video regions containing cilia, and optical flow fields are computed from consecutive frames as a compressed representation of temporal behavior. The predicted segmentation masks and optical flow entities are concatenated with the original video data as additional channels to each frame to form an augmented video. Each expanded video is fed frame-by-frame to the appearance module which utilizes a Variational Autoencoder (VAE) [KW19] to learn a compressed spatial representation for images of cilia. Videos are then embedded as sequences of points in the compressed latent space. The dynamics module employs another VAE to learn a representation from this compressed sequence, in order to reduce the amount of irrelevant information considered. If it were to instead train on the original video itself, the information would be too high-volume, potentially drowning out useful information in a sea of noise. This compressed sequence allows it to focus only on the motion of cilia. The dynamics module VAE is trained on potentially random subsequences of the embedded representations of video in order to assure that the temporal representation learned is adequately robust to reconstruct arbitrary parts of the sequence. Through this construction, we factor the representation of cilia into disentangled spatial and temporal components.

Data

Our data, obtained from the Quinn 2015 study, consist of nasal biopsy samples observed in patients with diagnosed ciliopathies and in healthy controls [QZD+15]. Nasal epithelial tissue was obtained from the inferior nasal turbinate using a Rhino-Pro curette, and cultured for three passages prior to recording. Grayscale video data was recorded for 1.25 seconds using a Phantom v4.2 high speed camera at 200 frames per second, resulting in 250 frames per sample. Recorded videos vary in dimension, ranging from 256 to 640 pixels on either axis. Segmentation masks used during the training of the preprocessing module were generated manually using ITK-SNAP, where each pixel is a binary value corresponding to whether the pixel contains cilia. Our dataset has a total of 325 sample videos, taken from Quinn 2015's cohort sampled at the University of Pittsburgh, and 230 ground-truth segmentation masks.

Because healthy cilia rhythmically beat at around 10-12Hz and our grayscale videos are recorded at 200 frames per second, there are approximately 17 frames per single ciliary beat cycle [QZD+15]. As such, we truncate our videos to 40 frames to capture at minimum 2 full beat cycles; the starting frame is randomly sampled. Because each video varies in dimensions, we

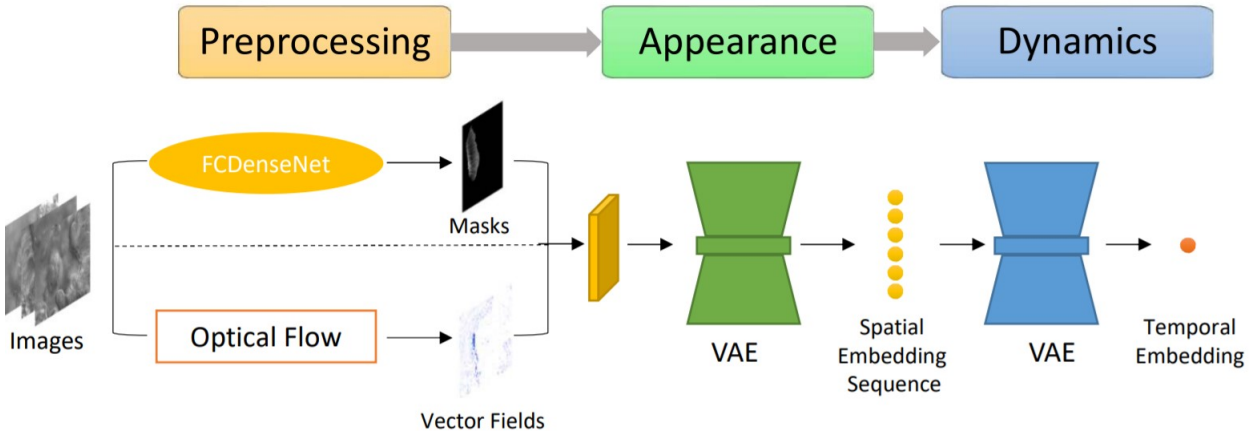


Fig. 1: *The proposed framework for creating a disentangled spatiotemporal representation*

obtain patches of size 128×128 as inputs to both the preprocessing and appearance modules. Instead of randomly sampling crops, we extract the first frame of the truncated video, and tile each frame-mask set such that no 128×128 patches overlap. The preprocessing module supplemented the 95 raw videos without corresponding ground-truth segmentation masks with segmentation masks predicted by a Fully Convolutional DenseNet.

Preprocessing

The preprocessing module primarily functions to generate segmentation masks that distinguish spatial regions containing cilia from background noise and supplement cilia data with measures of temporal behavior, such as optical flow and its derivative values.

Because we are interested in modelling the spatiotemporal behavior of *only* cilia, segmentation masks, which provide a direct mapping to pixels of interest within each frame, are critical within the appearance module to limit representation learning to cilia localities and ignore background noise. Although the end-to-end pipeline provides an unsupervised framework to represent and characterize complex and dynamic ciliary motion phenotypes, this module utilizes *supervised* segmentation to produce initial segmentation masks. Because we do not have ground-truth segmentation masks for every sample in our dataset, a supervised network allows us to augment our set such that each raw video has a corresponding segmentation mask to be used in subsequent modules. We draw upon prior supervised segmentation literature to implement FCDenseNet, a fully convolutional dense network that is able to leverage deep learning advantages without excessive parameters or loss of resolution. Each layer in a DenseNet is connected to every other layer in a feed-forward fashion; each layer takes the previous layers' feature maps as input, and its respective feature map is used by following layers. Fully Connected DenseNets (FCDenseNets) expand on this architecture with the principle goal of upsampling to recover input resolution [JDV+17]. Building a straightforward upsampling path requires multiplication of high-resolution feature maps, resulting in a computationally intractable number of feature maps. To mitigate this "feature explosion" issue, FCDenseNets upsample only the preceding dense block instead of upsampling all feature maps concatenated in previous layers. We modify and train a FCDenseNet to generate usable segmentation masks as input to the appearance module. Our architecture, shown in 2, consists of dense blocks, transition blocks, and skip connections totalling to 103 layers.

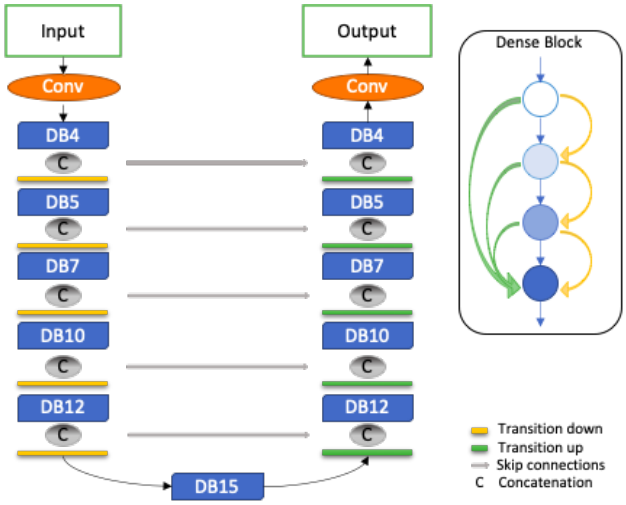


Fig. 2: *Fully Convolutional Dense Net with 103 layers*

Although we utilize a supervised segmentation network, we note that this is not necessary. We will be pursuing unsupervised methodologies with comparable efficacy, and chose the supervised network for the sake of creating an initial implementation and proof of concept

Since we aim to represent both spatial and temporal features, it is critical to obtain optical flow vector fields as a quantifiable proxy for ciliary movement. Two dimensional motion can be thought of as the projection of three dimensional motion on an image plane, relative to a visual sensor such as a camera or microscope. As such, optical flow represents the apparent motion of pixels within consecutive frames, relative to the visual sensor. To calculate pixel displacement, optical flow algorithms are contingent on several assumptions.

1) Brightness constancy assumes that a pixel's apparent intensity does not change between consecutive frames
2) Small motion assumes that pixels are not drastically displaced between consecutive frames
3) Spatial and temporal coherence assumes that a pixel's neighbors likely exhibit similar motion over gradual time

Solving these constraints yields a series of dense optical flow vector fields; each vector represents a pixel, and the magnitude

and direction of each vector signal the estimated pixel position in the following frame. We refer to Beauchemin and Barron [BB95] for detailed mathematical expression of optical flow derivation. Healthy cilia largely exhibit delicate textural behavior in which patches of cilia move synchronously, slowly, and within a set spatial region near cell boundaries. Additionally, our imaging modality allowed for consistent object brightness throughout sequences of frames. As such, we explored optical flow solutions that focus on brightness constancy, small motion, and spatial coherence systems of equations.

Our optical flow fields are computed using a coarse-to-fine implementation of Horn-Schunck's influential algorithm. Although we tested other methods, namely Farneback [Far03], Lucas-Kanade [LK81], and TV-L1 [SPMLF13], coarse-to-fine Horn-Schunck produced fields more robust to background movement. Horn-Schunck operates by firstly assuming motion smoothness between two frames; the algorithm then minimizes perceived distortions in flow by iteratively updating a global energy function [HS81]. The coarse-to-fine aspect transforms consecutive frames into Gaussian image pyramids; at each iteration, corresponding to levels in the Gaussian pyramids, an optical flow field is generated by Horn-Schunck, and then used to "warp" the images toward one another. This process is repeated until the two images converge. While Horn-Schunck has potential to be noise-sensitive due to its smoothness assumption, we observe that this is mitigated by the coarse-to-fine estimation and hyperparameter tuning. Additionally, we find that this estimation is more computationally and time efficient than its contemporaries.

For further insight into behavioral patterns, we extract first-order differential image quantities from our computed optical flow fields. Estimating linear combinations of optical flow derivatives results in orientation-invariant quantities: curl, deformation, and divergence [FK04]. Curl represents apparent rotation; each scalar in a curl field signaling the speed and direction of local angular movement. Deformation is the shearing about two different axes, in which one axis extends while the other contracts. Divergence, or dilation, is the apparent movement toward or away from the visual sensor, in which object size changes as a product of varied depth. Because our cilia data are captured from a top-down perspective without possibility of dilation, we limit our computation to curl and deformation, similar to Quinn 2011 [QFLC11].

Introduction To Autoencoders

Both the appearance and dynamics modules ultimately rely on a choice of a particular generative model. The chosen model greatly affects the rendered representation, and thus the efficacy of the entire pipeline. Our current choice of generative model is a VAE, an architecture that generates a low-dimensional representation of the data, parameterized as a probability distribution. A VAE can be considered a modified autoencoder (AE). A general AE attempts to learn a low-dimensional representation of the data by enforcing a so-called "bottleneck" in the network. This bottleneck is usually in the form of a hidden layer whose number of nodes is significantly smaller than the dimensionality of the input. The AE then attempts to reconstruct the original input using only this bottleneck representation. The idea behind this approach is that to optimize the reconstruction, only the most essential information will be maintained in the bottleneck, effectively creating a compressed, critical information based representation of the input data. The size of the bottleneck is a hyperparameter which determines how much of the data is compressed.

With this task in mind, an AE can be considered as the composition of two constituent neural networks: the encoder, and the decoder. Suppose that the starting dataset is a collection of n-dimensional points, $S \subset \mathbb{R}^n$, and we want the bottleneck to be of size l, then we can write the encoder and decoder as functions mapping between $\mathbb{R}^n$ and $\mathbb{R}^l$:

$$E_\theta : \mathbb{R}^n \to \mathbb{R}^l, \quad D_\theta : \mathbb{R}^l \to \mathbb{R}^n$$

The subscript θ denotes that these functions are constructed as neural networks parameterized by learanble weights θ. The encoder is tasked with taking the original data input and sending it to a compressed or *encoded* representation. The output of the encoder serves as the bottleneck layer. Then the decoder is tasked with taking this encoded representation and reconstructing a plausible input which could have been encoded to generate this representation, and thus is encouraged to become an approximate inverse of the encoder. The loss target of a AE is generally some distance function (not necessarily a metric) between items in the data space, which we denote as

$$d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}.$$

Given a single input $x \in S$, we then write the loss function as

$$L_\theta(x) = d(x, D_\theta(E_\theta(x)))$$

where a common choice for d is the square of the standard euclidean norm, resulting in

$$L_\theta(x) = \|x - D_\theta(E_\theta(x))\|^2.$$

The AE unfortunately is prone to degenerate solutions where when the decoder is sufficiently complex, rather than learning a meaningful compressed representation, it instead learns a hash of the input dataset, achieving perfect reconstruction at the expense of any generalizability. Notably, even without this extreme hash example, there is no restraint on continuity on the decoder, thus even if a point $z \in E(S) \subset \mathbb{R}^l$ in the latent space decodes into a nice, plausible data point in the original dataset, points close to z need not nicely decode.

The Variational Autoencoder

A VAE attempts to solve this problem by decoding neighborhoods around the encoded points rather than just the encoded points themselves. A neighborhood around a point $z \in \mathbb{R}^l$ is modeled by considering a multivariate gaussian distribution centered at $\mu \in \mathbb{R}^l$ with covariance $\Sigma \in \mathbb{R}^{l \times l}$. It often suffices to assert that the covariance be a diagonal matrix, allowing us to write $\Sigma = \text{diag}(\sigma)$ for some $\sigma \in \mathbb{R}^l$. While the decision to model neighborhoods via distributions deserves its own discussion and justification, it falls outside the scope of this paper and thus we omit the technical details while referring curious readers to [Doe16] for further reading. Instead, we provide a sort of rationalization of the conclusions of those discussions in the paragraphs that follow. While this is a little backwards, we find it does a better job of communicating the nature of the techniques to most audiences than does touring the complex mathematical underpinnings. The idea of modeling neighborhoods as distributions is implemented by changing the encoder to a new function

$$\tilde{E}_\theta : \mathbb{R}^n \to \mathbb{R}^l \times \mathbb{R}^l, \quad \tilde{E}_\theta : x \mapsto (\mu, \sigma)$$

where μ is the analog to the encoded z in the AE. However now we also introduce σ, which is the main diagonal of a covariance

matrix Σ, which determines how far, and in what direction, to randomly sample around the mean μ. What this means is after encoding, we no longer get a singular point, but a distribution modeling a neighborhood of points as promised. This distribution is referred to as the *posterior distribution* corresponding to x, written as $q(z|x) = \mathcal{N}(\mu, \Sigma)$. We sample from this posterior using the following construction

$$z \sim q_\theta(z|x) \iff z = \mu + \Sigma\varepsilon, \text{ where } \varepsilon \sim \mathcal{N}(0, I_l)$$

to ensure that we may complete backpropagation, since μ, σ are dependent on weights within the network. This is known as the reparameterization trick. Our modified loss is then

$$L_\theta(x) = \|x - D_\theta(z)\|^2.$$

Through this change, over the course of training we obtain a Monte Carlo estimation of the neighborhoods around the embedded points, encouraging continuity in their decoding. This result is still incomplete in that there's no guarantee that the decoder doesn't degenerate to setting σ arbitrarily close to zero, resulting in a slightly more complex AE. Thus we assert that if one were to sample from some predetermined *prior distribution* on the latent space, written as $p(z)$, then the sampled point can be reasonably decoded as a point in the starting data space. To break that down, this means that the portions of the latent space that our model should be best trained on should follow the prior distribution. A common choice for prior, due to simplicity, is the unit-variance Gaussian distribution. This is implemented by imposing a Kullback–Leibler Divergence (KL Divergence) loss between the posterior distributions (parameterized by our encoder via μ, σ) and the prior distribution (in this case $\mathcal{N}(0, I_l)$). Thus our final loss function is

$$L_\theta(x) = \|x - D_\theta(z)\|^2 + \mathrm{KL}(q_\theta(z|x) \,\|\, p(z)).$$

Now we finally have a vanilla VAE, wherein it can not only encode and decode the starting dataset, but it can also decode points in the latent space that it hasn't explicitly trained with (though with no strict promises on the resulting quality). Further improvements to the VAE framework have been made in recent years. To empower the decoder without introducing a significant number of parameters, we implement a spatial broadcast decoder (SBD), as outlined in [WMBL19]. To achieve greater flexibility in terms of the shape of the prior and posterior distributions, we employ the VampPrior in [TW17] with an added regularization term. Both these changes afford us greater flexibility and performance in creating a semantically meaningful latent space. The VampPrior is an alternative prior distribution that is constructed by aggregating the posteriors corresponding to K learned pseudo-inputs $\chi_1, \ldots, \chi_K$. The distribution is given by

$$p(z) = \frac{1}{K} \sum_i^K q(z|\chi_i)$$

This choice of prior optimizes the pipeline for downstream tasks such as clustering and phenotype discovery. We apply a regularization term to the loss to encourage that these pseudo-inputs look as though they could be reasonably generated by the starting dataset 3. Thus our loss becomes

$$\tilde{z}_i \sim q(z|\chi_i)$$

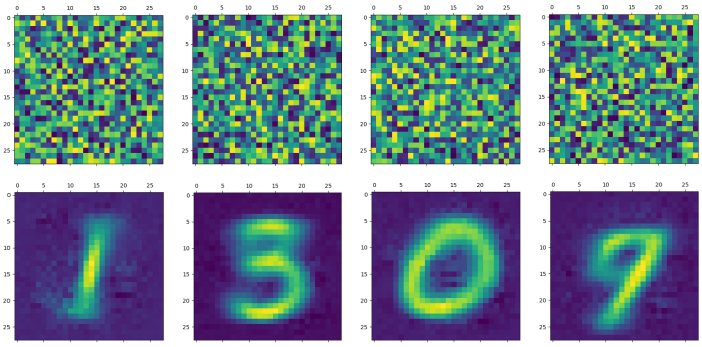


Fig. 3: *Pseudo-inputs of a VampPrior based VAE on MNIST without additional regularization term (top row), and with regularization term (bottom row)*

$$L_\theta(x) = \|x - D_\theta(z)\|^2 + \mathrm{KL}(q_\theta(z|x) \,\|\, p(z)) + \gamma \left(\sum_i^K \|\chi_i - D_\theta(\tilde{z}_i)\|^2 + \mathrm{KL}(q_\theta(z|\chi_i) \| p(z)) \right)$$

This has an immediate use in both clustering and semantic pattern discovery tasks. Rather than the embedding $E(S) \subset \mathbb{R}^l$ of the dataset being distributed as a unit gaussian, it is distributed as a mixture of gaussians, with each component being a posterior of a pseudo-input. Consequently, the pseudo-inputs create notable and calculable clusters, and the semantic significance of the clusters can be determined, or at least informed, by analyzing the reconstruction of the pseudo-input responsible for that posterior distribution.

Appearance

The appearance module's role is to learn a sufficient representation so that, frames are reconstructed accurately on an individual basis, and that spatial differences of frames over time is represented with a meaningful sequence of points in the latent space. The latter is the core assumption of the dynamics module.

The appearance module is designed to work with generalized videos, regardless of specific application. Specifically it is designed to take as input singular video frames, augmented with the information generated during the preprocessing phase, including optical flow quantities such as curl. These additional components are included as additional channels concatenated to the starting data, and thus is readily expandable to suit whatever augmented information is appropriate for a given task. In the case of our particular problem, one notable issue is that cilia occupy a small portion of the frame as shown in Figure 6, and thus, the contents of the images that we are interested in exist in some subspace that is significantly smaller than the overall data space. This can result in problems where the neural network optimizes components such as background noise and image artifacts at the expense of the clinically critical cilia information. To remedy this, we leverage the segmentation masks created during the preprocessing phase to focus the network on only the critical portions of the image.

To that effect, we mask the augmented frame data—the raw images concatenated with additional information such as optical flow quantities—using the segmentation masks and train the network on these masked quantities. Mathematically we refer to a single augmented frame with k channels as f, a doubly-indexed collection of vectors, writing the channel information of pixel (i, j) as $f_{i,j} \in \mathbb{R}^k$. We similarly write the generated segmentation mask

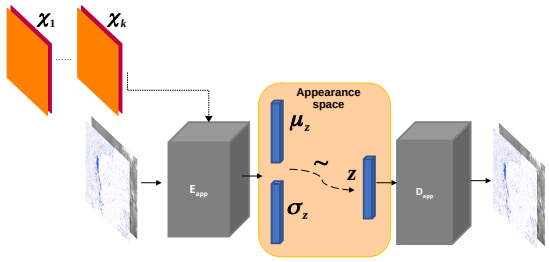


Fig. 4: *The Appearance Module pipeline*

m as a doubly-indexed collection of scalars with $m_{i,j} \in [0,1] \subset \mathbb{R}$, then we construct the augmented frame

$$\tilde{f}_{i,j} := f_{i,j} \cdot m_{i,j}$$

The appearance module ultimately embeds a segmented region of cilia observed in a single frame into what we refer to as the appearance latent space. Due to the temporally static nature of individual frames, this latent space is an encoded representation of *only* the spatial information of the underlying processed data. This spatial information includes aspects such as the shape and distribution of cilia along cells, as well as factors such as their length, orientation and overall shape. These spatial features can be then used in downstream tasks such as phenotype discovery, by drawing a strong connection between apparent patterns in the appearance latent space as and semantically meaningful patterns in the underlying data as well.

Our proposed model for the appearance module uses a variation of ResNet [HZRS15] as an encoder, while employing SBD in the decoder, as well as an upsampling, ResNet-like network in the decoder. Figure 4 shows the training pipeline for the appearance module, the encoder E_{app} is the neural network implementing the variational distribution $q(z|x)$, by estimating parameters to a normal distribution given a certain input frame $x = f$. $q(z|x)$ is therefore $\mathcal{N}(\mu_z, diag(\sigma_z))$ where μ_z and σ_z are the mean and standard deviation vectors of a normal distribution estimated given a certain input frame, or a pseudo-input frame. The decoder D_{app} is trained to reconstruct the input from a sampled $z \sim \mathcal{N}(\mu_z, diag(\sigma_z))$, by minimizing the $L2$ loss between the input and the reconstructed output. The pseudo-inputs $\chi_{1..k}$ are only used during the training, to enforce the prior constraint through a Monte Carlo estimation of the KL divergence, as mentioned earlier.

Dynamics

While the appearance module handles representing the video frames individually under a generative model, the dynamics module is where the temporal behavior is represented. We propose a VAE generative seq2seq module that consists of both an encoder and a decoder to embed the temporal dynamics in a latent semantic space for motion patterns (dynamics). The encoder handles embedding the dynamics of the observed video frames (input) into a latent vector w in the dynamics semantic space $\mathbb{R}^{d_{\text{dyn}}}$. This vector w encodes the dynamics of the video subsequence observed by the encoder. The decoder, then, is able to extrapolate the video into future time-steps by unrolling a sampled latent vector w from the dynamics space into a sequence of vectors $c_{1..k}$. These vectors are not the extrapolated sequence themselves, but instead represent a sequence of changes to be made on a supplied appearance vector $\hat{z}_0$. This vector serves as an initial frame—a starting point for extrapolation—and can be any frame from the video since the vector w encodes the dynamics of the *entire* video. Applying this sequence of change vectors to the initial appearance vector one-by-one, using an aggregation operator $\phi(z,c)$, which could be as simple as vector addition, results in a sequence of appearance vectors $\hat{z}_{1..k}$ which represent the extrapolated sequence. This sequence can then be decoded back into video frames through the decoder of the appearance module D_{app}.

Since the encoder and the decoder of the dynamics module need to process sequences of vectors, they are modeled using a Gated Recurrent Unit (GRU) [CvMG+14] and an LSTM unit, respectively. They are types of RNN with unique architectures that allow them to handle longer sequences of data than a generic RNN could. A GRU cell operates on an input vector x_t, and a hidden state vector s^t at a certain time-step t. Applying a GRU step results in an updated state vector s^{t+1}. An LSTM cell is similar, but it also has an additional output state h^t that gets updated as well like the hidden state.

Figure 5 depicts the pipeline of the proposed dynamics module, showing the encoder steps, sampling from the dynamics space, and the decoder steps. The dynamics encoder GRU, E_{dyn}, starts from a blank state vector $s^0_{\text{enc}} = 0$ that updates every time the appearance vector of the next video frame is fed-in. After feeding in the appearance vector of the final input frame z_n, the state vector s^n_{enc} would encompass information about the motion patterns in the observed video frames $z_{1..n}$, and would then constitute a latent vector in the dynamics semantic space $w = s^n$.

The dynamics decoder LSTM D_{dyn} starts from a latent dynamics vector as its hidden state $s^0_{\text{dec}} = w$, a blank output state vector $h^0_{\text{dec}} = 0$ and an initial supplied appearance vector to act as the beginning output frame. Note that this supplied vector can be any point but the last within the original input sequence, thus we set $\hat{z}_0 = z_i$ for some $i \in \{1,\ldots,n-1\}$. Applying each step results in a change vector $c^{t+1} = h^{t+1}$ (output state vector), that gets applied to the most recent appearance vector in the output sequence to predict the next appearance vector $\hat{z}_{t+1} = \phi(\hat{z}_t, c^{t+1})$, which in turn is used as an input vector to the next LSTM step. The sequence of predicted appearance vectors are then passed through the appearance decoder $D_{\text{app}}(\hat{z}_1), \ldots, D_{\text{app}}(\hat{z}_k)$, to generate back the video frames. During training time, an L_2 loss is minimized on the predicted k points in the appearance latent space and the true ones.

A prior constraint is imposed on the encoder's output, as per the VAE formulation. Therefore, the size of the state vector of the encoder is $2d_{\text{dyn}}$, composed of both μ_w, and σ_w, such that $w \sim \mathcal{N}(\mu_w, \text{diag}(\sigma_w))$. The prior loss then becomes $\text{KL}(\mathcal{N}(\mu_w, \text{diag}(\sigma_w)) \,||\, \mathcal{N}(0, I))$ and is minimized throughout the training.

It is important to note that the appearance module and the dynamics module are decoupled, such that sampling a different vector w from the dynamics latent space results in different motion dynamics in the extrapolated sequence of video frames despite starting from the same initial supplied frame. As is the case when supplying a different initial output frame as well. To reinforce that, after encoding an input sequence into a dynamics latent vector w, multiple sequences of $k+1$ frames are sampled uniformly from the same training video, where each generated sequence is set to extrapolate from its first frame, and the same dynamics vector w. The $L2$ loss between the extrapolated frames and the remaining k frames in each sequence is minimized with backpropagation.

To summarize, encoder of the dynamics module is trained to extract the motion dynamics from the appearance vectors of

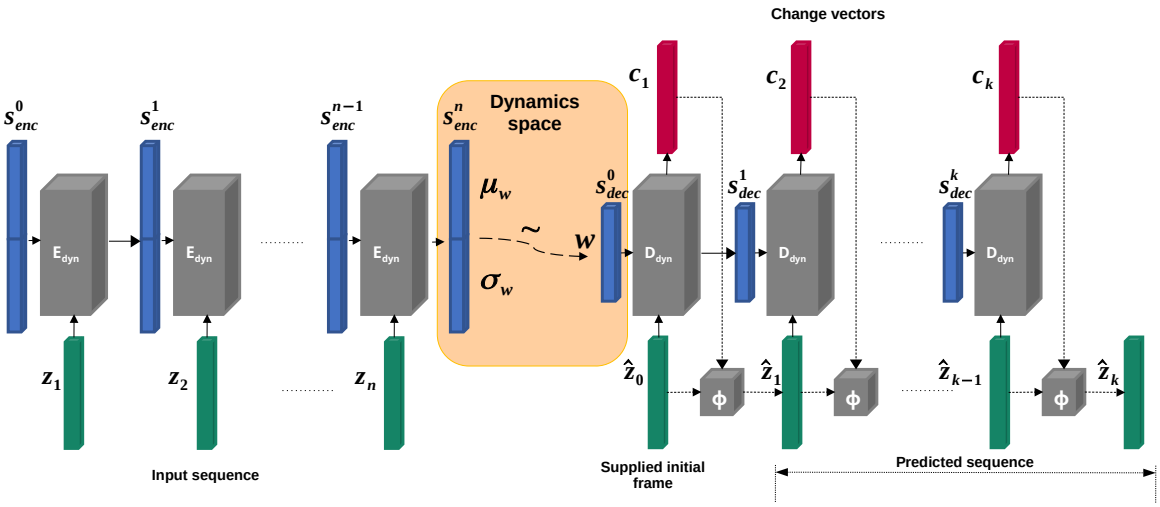


Fig. 5: The Dynamics Module pipeline

a sequence of video frames, and embeds them in a semantic space representing all possible cilia motion patterns. The decoder applies motion patterns from a sampled dynamics vector to a given starting frame, and predicts the appearance vectors to future frames.

Results

The data used for the segmentation task consists of 223 corresponding sets of ground truth masks and high-speed digital microscopy video data. The ground truth masks were manually generated to represent regions of cilia, and the video contains time-series differential image contrast grayscale frames. Each model trained is evaluated by testing intersection over union (IoU), testing precision, and testing accuracy. For every mask generated by FCDN-103, IoU computes the region of overlap between predicted pixels containing cilia and ground truth pixels containing cilia over the joint regions of either prediction or ground truth that contain cilia. Although IoU is typically a superior metric for segmentation evaluation, FCDN-103 is optimized with the goal of minimizing type II error or the presence of false positives because the output masks will be used to narrow representation learning to our region of interest. Thus, we aim to produce segmentation masks with high precision that exclusively identify regions of cilia containing minimal background scene.

We train our FCDN-103 model, written in PyTorch [PGM+19], with an Adam optimizer and cross-entropy loss on one NVIDIA Titan X GPU card. We split our data to consist of 1785 training patches and 190 testing patches. Throughout training and tuning, we experiment with several parameters: standard parameters such as batch size, learning rate, and regularization parameters such as learning rate decay, weight decay, and dropout. We observe optimal performance after 50 epochs, 14 patches per batch, learning rate of 0.0001, learning rate decay of 0.0, and weight decay of 0.0001. This model achieves 33.06% average testing IOU, and 53.26% precision. Figure 6 shows two examples of 128 x 128 test patches with their corresponding ground truth mask (middle) and FCDN-103 generated mask (right); the predicted masks cover more concise areas of cilia than the ground

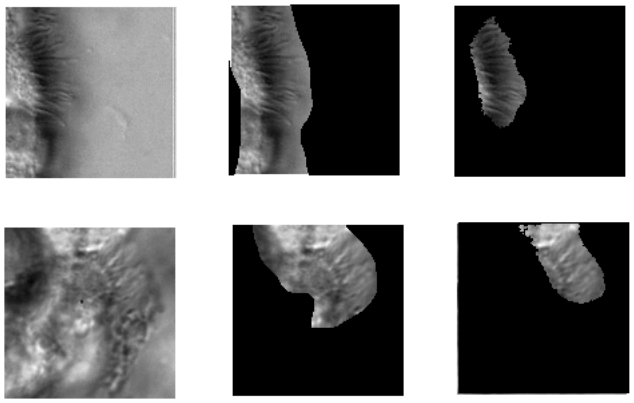


Fig. 6: Segmentation examples from left to right: raw test frame, frame overlain with ground truth segmentation mask, frame overlain with FCDN-103 predicted segmentation maskline

truths and ignore the background in entirety. Previously, Lu 2018 implement a Fully Convolutional DenseNet with 109 layers in a tiramisu architecture trained on ciliary data [LMZ+18]; FCDN-103 achieves an average of 88.3% testing accuracy, outperforming Lu 2018's FCDN-109 by two percentage points.

Curl and deformation fields are extracted from the generated optical flow fields using SciPy's signal and ndimage packages [VGO+20]. Figure 7 shows an example of healthy cilia and its mid-cycle optical flow where vector magnitude corresponds to color saturation; we can reasonably assume that the primary region of movement within optical flow fields will contain healthy cilia. While optical flow fields can potentially provide information on cilia location, we avoid solely using optical flow fields to generate segmentation masks due to the presence of dyskinetic cilia. Identifying stationary cilia is a crucial step in learning ciliary motion phenotype. However, it is possible that optical flow provides insight into both ciliary location and temporal behavior.

During optimization of the appearance module, we observe that cilia do not tend to exhibit a large degree of spatial differences

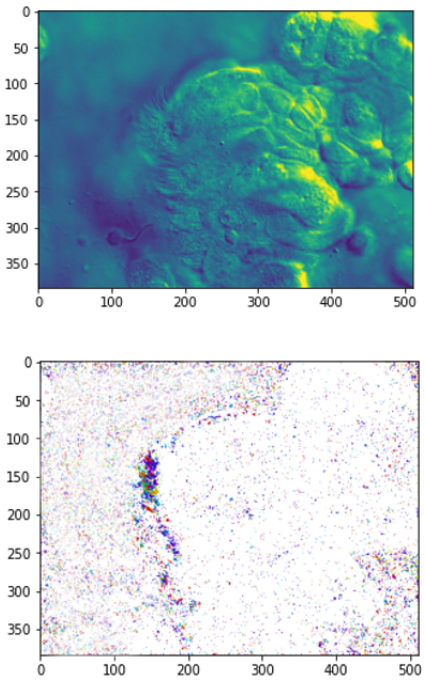


Fig. 7: *Raw imagery and corresponding optical flow visualization*

over time, thus rather than processing every frame of the dataset, we used the NumPy [vdWCV11] library to efficiently sample a fixed number of frames from each video. For testing purposes, we set the number of sampled frames to 40. We sample these frames uniformly throughout the video to ensure that both high-frequency (e.g. cilia beats) and low-frequency (e.g. cell locomotion) spatial changes are represented to ensure that we train on a sufficiently varied base of spatial features.

The entirety of the appearance module's architecture was written using PyTorch. The encoder is a composition of residual blocks, with pixel-wise convolutions and maxpooling operations between them to facilitate channel shuffling and dimensionality reduction respectively, connecting to a fully-connected layer which represents the means and log-variances along each axis of the latent space. We use log-variances instead of the usual standard deviation, or even variance, to guarantee numerical stability, make subsequent calculations such as KL divergence easier, and reduce the propensity for degenerate distributions with variances that approach 0. Since we use a modified VampPrior, the KL Divergence is between a single Gaussian, the posterior, and a mixture of Gaussians, the prior, and thus intractable. In order to estimate this, we employ a Monte Carlo estimation technique, manually calculate the difference in log-probabilities for each distribution at every pass of the loss function, asserting that throughout training these values approximate the ideal KL Divergence. All figures were generated using Matplotlib [Hun07]. The current project can be found at our github repository.

Conclusion

While the initial task of this model was to represent cilia, it also serves as a general framework that is readily extensible to almost any task that involves the simultaneous, yet separate, representation of spatial and temporal components. The specific aim of this project was to develop separate, usable tools which sufficiently accomplish their narrow roles and to integrate them together to offer a more meaningful understanding of the overall problem. While we are still in the early phases of evaluating the entire integrated pipeline as a singular solution, we have demonstrated early successes with the preprocessing module, and have situated the appearance and dynamics modules in the context of modern machine learning approaches well enough to justify further exploration.

Further Research

This generative framework is a foundational element of a much larger project: the construction of a complete library of ciliary motion phenotypes for large-scale genomic screens, and the development of a comprehensive and sophisticated analytics toolbox. The analytics toolbox is intended to be used by developmental and molecular biologists in research settings, as well as clinicians in biomedical and diagnostic settings. By packaging this framework in an easy-to-use open source toolbox, we aim to make sophisticated generative modeling of ciliary motion waveforms available to researchers who do not share our machine learning backgrounds. This pipeline will also serve as a basis and back-end for an exploration into the realm of collaborative, crowd-driven data acquisition and processing in the form of a user-friendly web tool.

More research should also be done to the implementations of each module, and namely their codependencies. For example, how do the quality of segmentation masks in the preprocessing module affect the quality of spatial representation, and consequently dynamic representation? Is there virtue in allowing partial entanglement between the appearance and dynamics module to optimize their joint representation? Can the learned spatial representation influence and inform the preprocessing module in a meaningful way? We hope to explore these questions, and many more, in the near future.

We also encourage the expansion and application of this framework to various other problem contexts. The modular approach to its design ensures portability and adaptability to other projects. The fact that the dynamics module is designed to operate within the abstract latent space of the appearance module means that the appearance module acts as a buffer or converter between the concrete data and the temporal analysis. Consequently, when applying the framework to new projects, only the appearance module need be altered, while the preprocessing module may optionally be adapted or entirely dropped, and the dynamics module preserved.

One example task this pipeline could be adapted to is that of RNA folding analysis. The study of RNA folding patterns is essential in areas such as drug development. One way to model RNA folding is to consider a strand of RNA as a partially-connected point cloud, tracked through time. In this case, the preprocessing module may be forgone, and altering the appearance encoder/decoder to a generic architecture compatible with point clouds, e.g. a geometric neural network or GCNN is all that is necessary. The dynamics module could be readily applied without significant changes.

Acknowledgements

This study was supported in part by Google Cloud. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research. This study was supported in part by NSF CAREER #1845915.

REFERENCES

[BB95] S. S. Beauchemin and J. L. Barron. The computation of optical flow. *ACM Computing Surveys (CSUR)*, 27(3):433–466, September 1995. URL: http://dl.acm.org/doi/10.1145/212094.212141, doi:10.1145/212094.212141.

[BMO17] Ximena M. Bustamante-Marin and Lawrence E. Ostrowski. Cilia and mucociliary clearance. *Cold Spring Harbor perspectives in biology*, 9(4), 2017. doi:10.1101/cshperspect.a028241.

[CA17] ANDREEA CATANA and ADINA PATRICIA APOSTU. The determination factors of left-right asymmetry disorders- a short review. *Clujul Medical*, 90(2):139–146, 2017. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5433564/, doi:10.15386/cjmed-701.

[CvMG+14] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014. arXiv:1406.1078, doi:10.3115/v1/D14-1179.

[DCWS03] Gianfranco Doretto, Alessandro Chiuso, Ying Nian Wu, and Stefano Soatto. Dynamic textures. *International Journal of Computer Vision*, 51(2):91–109, Feb 2003. URL: https://doi.org/10.1023/A:1021669406132, doi:10.1023/A:1021669406132.

[Doe16] Carl Doersch. Tutorial on variational autoencoders, 2016. arXiv:1606.05908.

[Far03] Gunnar Farnebäck. Two-Frame Motion Estimation Based on Polynomial Expansion. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Josef Bigun, and Tomas Gustavsson, editors, *Image Analysis*, volume 2749, pages 363–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/3-540-45103-X_50, doi:10.1007/3-540-45103-X_50.

[FK04] Shih Ching Fu and Peter Kovesi. Extracting Differential Invariants of Motion Directly From Optical Flow. In *13th School of Computer Science & Software Engineering Research Conference*, 2004. doi:10.1.1.185.1179.

[FL12] Thomas W Ferkol and Margaret W Leigh. Ciliopathies: the central role of cilia in a spectrum of pediatric disorders. *The Journal of pediatrics*, 160(3):366, 2012. doi:10.1016/j.jpeds.2011.11.024.

[GSC99] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2, 1999. doi:10.1162/089976600300015015.

[GZT+14] Andrea S Garrod, Maliha Zahid, Xin Tian, Richard J Francis, Omar Khalifa, William Devine, George C Gabriel, Linda Leatherbury, and Cecilia W Lo. Airway ciliary dysfunction and sinopulmonary symptoms in patients with congenital heart disease. *Annals of the American Thoracic Society*, 11(9):1426–1432, 2014. doi:10.1513/AnnalsATS.201405-222OC.

[HGGRD99] Els Houtmeyers, Rik Gosselink, Ghislaine Gayan-Ramirez, and Marc Decramer. Regulation of mucociliary clearance in health and disease. *European Respiratory Journal*, 13(5):1177–1188, 1999. doi:10.1034/j.1399-3003.1999.13e39.x.

[HLW16] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. URL: http://arxiv.org/abs/1608.06993, arXiv:1608.06993, doi:10.1109/CVPR.2017.243.

[HS81] Berthold K P Horn and Brian G Schunck. Determining Optical Flow. *Artificial Intelligence*, 17:185–203, 08 1981. doi:10.1016/0004-3702(81)90024-2.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, 2007.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL: http://arxiv.org/abs/1512.03385, arXiv:1512.03385, doi:10.1109/CVPR.2016.90.

[JDV+17] Simon Jegou, Michal Drozdzal, David Vazquez, Adriana Romero, and Yoshua Bengio. The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation. *arXiv:1611.09326 [cs]*, October 2017. arXiv: 1611.09326. URL: http://arxiv.org/abs/1611.09326, doi:10.1109/CVPRW.2017.156.

[KSC17] Celine Kempeneers, Claire Seaton, and Mark A Chilvers. Variation of ciliary beat pattern in three different beating planes in healthy subjects. *Chest*, 151(5):993–1001, 2017. doi:10.1016/j.chest.2016.09.015.

[KW19] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019. URL: http://arxiv.org/abs/1906.02691, arXiv:1906.02691, doi:10.1561/2200000056.

[LK81] Bruce Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision (ijcai). volume 81, 04 1981.

[LMZ+18] Charles Lu, M. Marx, M. Zahid, C. W. Lo, Chakra Chennubhotla, and Shannon P. Quinn. Stacked neural networks for end-to-end ciliary motion analysis. *CoRR*, abs/1803.07534, 2018. URL: http://arxiv.org/abs/1803.07534, arXiv:1803.07534.

[OCH+07] Christopher O'Callaghan, Mark Chilvers, Claire Hogg, Andrew Bush, and Jane Lucas. Diagnosing primary ciliary dyskinesia, 2007. doi:10.1136/thx.2007.083147.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[QFLC11] Shannon Quinn, Richard Francis, Cecilia Lo, and Chakra Chennubhotla. Novel use of differential image velocity invariants to categorize ciliary motion defects. In *Proceedings of the 2011 Biomedical Sciences and Engineering Conference: Image Informatics and Analytics in Biomedicine*, pages 1–4, Knoxville, TN, USA, March 2011. IEEE. URL: http://ieeexplore.ieee.org/document/5872328/, doi:10.1109/BSEC.2011.5872328.

[QZD+15] Shannon P Quinn, Maliha J Zahid, John R Durkin, Richard J Francis, Cecilia W Lo, and S Chakra Chennubhotla. Automated identification of abnormal respiratory ciliary motion in nasal biopsies. *Science translational medicine*, 7(299):299ra124–299ra124, 2015. doi:10.1126/scitranslmed.aaa1233.

[RWH+14] Johanna Raidt, Julia Wallmeier, Rim Hjeij, Jörg Große Onnebrink, Petra Pennekamp, Niki T Loges, Heike Olbrich, Karsten Häffner, Gerard W Dougherty, Heymut Omran, et al. Ciliary beat pattern and frequency in genetic variants of primary ciliary dyskinesia. *European Respiratory Journal*, 44(6):1579–1588, 2014. doi:10.1183/09031936.00052014.

[SPMLF13] Javier Sánchez Pérez, Enric Meinhardt-Llopis, and Gabriele Facciolo. TV-L1 Optical Flow Estimation. *Image Processing On Line*, 3:137–150, July 2013. URL: http://www.ipol.im/pub/art/2013/26/?utm_source=doi, doi:10.5201/ipol.2013.26.

[SS90] Peter Satir and Michael A Sleigh. The physiology of cilia and mucociliary interactions. *Annual review of physiology*, 52(1):137–155, 1990. doi:10.1146/annurev.ph.52.030190.001033.

[TW17] Jakub M. Tomczak and Max Welling. VAE with a vampprior. *CoRR*, abs/1705.07120, 2017. URL: http://arxiv.org/abs/1705.07120, arXiv:1705.07120.

[vdWCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011. URL: http://arxiv.org/abs/1102.1523, arXiv:1102.1523, doi:10.1109/MCSE.2011.37.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python.

Nature Methods, 17:261–272, 2020. doi:https://doi.org/10.1038/s41592-019-0686-2.

[WMBL19] Nicholas Watters, Loic Matthey, Christopher P. Burgess, and Alexander Lerchner. Spatial broadcast decoder: A simple architecture for learning disentangled representations in vaes, 2019. URL: http://arxiv.org/abs/1901.07017, arXiv:1901.07017.