Proceedings of the 20th Python in Science Conference

July 12 - July 18 • Austin, Texas

PROCEEDINGS OF THE 20TH PYTHON IN SCIENCE CONFERENCE

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

SciPy 2021
Austin, Texas
July 12 - July 18, 2021

ISSN:2575-9752
https://doi.org/10.25080/majora-1b6fd038-02b

ORGANIZATION

Conference Chairs

JONATHAN GUYER, NIST
CORRAN WEBSTER, Enthought, Inc.

Program Chairs

MATT HABERLAND, Cal Poly
MADICKEN MUNK, University of Illinois

Communications

MATT DAVIS, Populus
DAVID NICHOLSON, Embedded Intelligence

Birds of a Feather

JULIE HOLLEK, Mozilla
NADIA TAHIRI, Université de Montréal

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

Tutorials

MIKE HEARNE, USGS
ERIC OLSEN, Enthought, Inc.
SERAH RONO, The Carpentries

Sprints

JUAN NUNEZ-IGLESIAS, Monash University

Diversity

CELIA CINTAS, IBM Research Africa
GUEN PRAWIROATMODJO, Microsoft Quantum

Activities

PAUL ANZEL, HEB
INESSA PAWSON, Albus Code

Sponsors

KRISTEN LEISER, Enthought, Inc.

Financial

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

Logistics

KRISTEN LEISER, Enthought, Inc.

Proceedings Reviewers

Adam Thompson
Alberto Antonietti
Anirudh Acharya
Brian Gue
Chiara Marmo
Chris Calloway
David Nicholson
David Shupe
Dillon Niederhut
Fatma Tarlaci
Heather Schovanec
Jyotika Singh
Kelvin Lee
Kirtan Dave
Kyle Penner
Matthew Feickert
Meghann Agarwal
Nadia Tahiri
Nuno Castro
Richard Liaw
Ryan Bunney
Shubham Sharma
Scott Sievert
Stéfan van der Walt
Tetsuo Koyama
Tyler Skluzacek
Tzu-Chi Yen

Accepted Talk Slides

It's Time for the Atmospheric Science Community to ACT Together, Adam Theisen
doi.org/10.25080/majora-1b6fd038-020
Adopting static typing in scientific projects, Predrag Gruevski, and Colin Carroll
doi.org/10.25080/majora-1b6fd038-021
cuCIM - A GPU image I/O and processing library, Gregory R. Lee, and Gigon Bae, and Benjamin Zaitlen, and John Kirkham, and Rahul Choudhury
doi.org/10.25080/majora-1b6fd038-022
Distributed statistical inference with pyhf powered by funcX, Matthew Feickert
doi.org/10.25080/majora-1b6fd038-023

Accepted Posters

Towards a Scientific Workflow Description: a yt Project Prototype for Interdisciplinary Analysis, Samantha Walkow, and Dr. Chris Havlin, and Dr. Matthew Turk, and Dr. Corentin Cadiou
doi.org/10.25080/majora-1b6fd038-017
Using Python for Analysis and Verification of Mixed-mode Signal Chains for Analog Signal Acquisition, Mark Thoren, and Cristina Suteu
doi.org/10.25080/majora-1b6fd038-018
Speeding Up Molecular Dynamics Trajectory Analysis with MPI Parallelization, Edis Jakupovic, and Oliver Beckstein
doi.org/10.25080/majora-1b6fd038-019
Social Media Analysis using Natural Language Processing Techniques, Jyotika Singh
doi.org/10.25080/majora-1b6fd038-01a
Programmatically Identifying Cognitive Biases Present in Software Development, Amanda E. Kraft, and Matthew Widjaja, and Trevor M. Sands, and Brad J. Galego
doi.org/10.25080/majora-1b6fd038-01b
Visualize 3D scientific data in a Pythonic way like matplotlib, Tetsuo Koyama
doi.org/10.25080/majora-1b6fd038-01c
causal-curve: tools to perform causal inference given a continuous treatment, Roni Kobrosly
doi.org/10.25080/majora-1b6fd038-01d
SciPy 2021: An Accurate Implementation of the Studentized Range Distribution for Python, Samuel Wallan, and Dominic Chmiel, and Matt Haberland
doi.org/10.25080/majora-1b6fd038-01e
Cell Tracking in 3D using Deep Learning Segmentations, Varun Kapoor, and Claudia Carabana
doi.org/10.25080/majora-1b6fd038-01f

SciPy Tools Plenaries

Awkward Array, Jim Pivarski
doi.org/10.25080/majora-1b6fd038-024
SciPy Tools Plenary on Matplotlib, Elliott Sales de Andrade
doi.org/10.25080/majora-1b6fd038-025
NumPy – Annual Update, Inessa Pawson
doi.org/10.25080/majora-1b6fd038-026
SciPy Tools Plenary: Jupyter Updates, Isabela Presedo-Floyd, and Matthias Bussonnier
doi.org/10.25080/majora-1b6fd038-027
Scientific Python Ecosystem Coordination, K. Jarrod Millman, and Stéfan van der Walt
doi.org/10.25080/majora-1b6fd038-028
SciPy: SciPy 2021 Tools Track, Pamphile T. Roy
doi.org/10.25080/majora-1b6fd038-029
SciPy Tools Plenary: scikit-image annual update, Gregory R. Lee
doi.org/10.25080/majora-1b6fd038-02a

Lightning Talks

Social Media Analysis using Natural Language Processing Techniques, Jyotika Singh
doi.org/10.25080/majora-1b6fd038-015
seaborn-image : image data visualization in Python, Sarthak Jariwala
doi.org/10.25080/majora-1b6fd038-016

Scholarship Recipients

Ahmed Abdelazeem Adam,
Jishan Ahmed,
Erick Faria,
T.S.Sachin Venkatesh,
Alessandro Lodi,
Pablo Adrian Toledo Margalef,
Moinak Bose,
Vinícius Salazar,
Vinicius Cerutti,
Ana Krelling,
Sai Santhosh Akavaram,
Idris Zakariyya,
Mauricio Ernesto Rodríguez Alas,
Shubham Sharma,
Gajendra Deshpande,
Israel Adeh-Aboh,
Serge Guelton,
Samuel Damilola Obafisoye,
Akihiro Nitta,
Francesco Andreuzzi,
Laura Gutierrez Funderburk,
Appah Akuamoah Adarkwah,
Neha Gupta,
Eduardo Destefani Stefanato,
Milan Marlon McGraw,
Wamba Tchinda Claudin,
Vinicius Cerutti,
Oluwakayode Olamoyegun,
Tyler Newton,
Kaira Paul,
Annajiat Alim Rasel,
Ernest Arko,
Emaude Altema,
Eyal Kazin,
Nurul Farahain Mohammad,
Aman Goel,
Victoria Sayo Turner,

CONTENTS

How PDFrw and fillable forms improves throughput at a Covid-19 Vaccine Clinic

Haw-minn Lu‡*, José Unpingco‡

Abstract—PDFrw was used to prepopulate Covid-19 vaccination forms to improve the efficiency and integrity of the vaccination process in terms of federal and state privacy requirements. We will describe the vaccination process from the initial appointment, through the vaccination delivery, to the creation of subsequent required documentation. Although Python modules for PDF generation are common, they struggle with managing fillable forms where a fillable field may appear multiple times within the same form. Additionally, field types such as checkboxes, radio buttons, lists and combo boxes are not straightforward to programmatically fill. Another challenge is combining multiple *filled* forms while maintaining the integrity of the values of the fillable fields. Additionally, HIPAA compliance issues are discussed.

Index Terms—acrobat documents, form filling, HIPAA compliance, COVID-19

Introduction

The coronavirus pandemic has been one of the most disruptive nationwide events in living memory. The frail, vulnerable, and elderly have been disproportionately affected by serious hospitalizations and deaths. Notwithstanding the amazing pace of vaccine development, logistical problems can still inhibit large-scale vaccine distribution, especially among the elderly. Vaccination centers typically require online appointments to facilitate vaccine distribution by State and Federal governments, but many elderly do not have Internet access or know how to make online appointments, or how to use online resources to coordinate transportation to and from the vaccination site, as needed.

As a personal anecdote, when vaccinations were opened to all aged 65 and older, one of the authors tried to get his parents vaccinated and discovered that the experience documented here [Let21] was unfortunately typical and required regularly pinging the appointment website for a week to get an appointment. However, beyond persistence, getting an appointment required monitoring the website to track when batches of new appointments were released --- all tasks that require an uncommon knowledge of Internet infrastructure beyond most patients, not just the elderly.

To help San Diego County with the vaccine rollout, the Gary and Mary West PACE (WestPACE) center established a pop-up point of distribution (POD) for the COVID-19 vaccine [pre21] specifically for the elderly with emphasis on those who are most vulnerable. The success in the POD was reported in the local news media [Lit21] [Col21] and prompted the State of California to ask WestPACE's sister organization (the Gary and Mary West Health Institute) to develop a playbook for the deploying a pop-up POD [pod21].

This paper describes the logistical challenges regarding the vaccination rollout for WestPACE and focuses on the use of Python's `PDFrw` module to address real-world sensitive data issues with PDF documents.

This paper gives a little more background of the effort. Next the overall infrastructure and information flow is described. Finally, a very detailed discussion on the use of python and the `PDFrw` library to address a major bottleneck and volunteer pain point.

Background

WestPACE operates a Program of All-Inclusive Care for the Elderly (PACE) center which provides nursing-home-level care and wrap-around services such as transportation to the most vulnerable elderly. To provide vaccinations to WestPACE patients as quickly as possible, WestPACE tried to acquire suitable freezers (some vaccines require special cold storage) instead of waiting for San Diego County to provide them; but, due to high-demand, acquiring a suitably-sized freezer was very problematic. As a pivot, WestPACE opted to acquire a freezer that was available but with excess capacity beyond what was needed for just WestPACE, and then collaborated with the County to use this excess capacity to establish a walk-up vaccination center for all San Diego senior citizens, in or out of WestPACE.

WestPACE coordinated with the local 2-1-1 organization responsible for coordination of community health and disaster services. The 2-1-1 organization provided a call center with in-person support for vaccine appointments and transportation coordination to and from WestPACE. This immediately eased the difficulty of making online appointments and the burden of transportation coordination. With these relationships in place, the vaccination clinic went from concept to active vaccine distribution site in about two weeks resulting in the successful vaccination of thousands of elderly.

Although this is a technical paper, this background describes the real impact technology can make in the lives of the vulnerable and elderly in society in a crisis situation.

Infrastructure

The goal of the WestPACE vaccine clinic was to provide a friendly environment to vaccinate senior citizens. Because this was a non-profit and volunteer effort, the clinic did not have any pre-existing

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

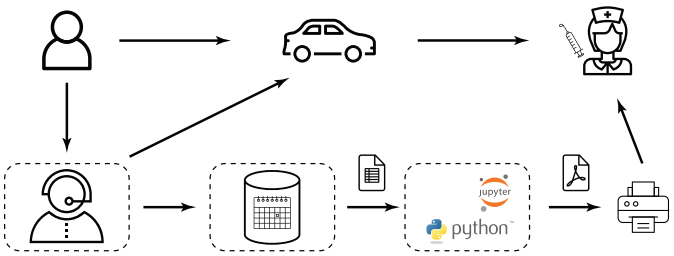

Fig. 1: *Vaccination Pipeline*

record management practices with corresponding IT infrastructure to handle sensitive health information according to Health Insurance Portability and Accountability Act (HIPAA) standards. One key obstacle is paperwork for appointments, questionnaires, consent forms, and reminder cards (among others) that must be processed securely and at speed, given the fierce demand for vaccines. Putting the burden of dealing with this paperwork on the patients would be confusing for the patient and time-consuming and limit the overall count of vaccinations delivered. Thus, the strategy was to use electronic systems to handle Protected Health Information (PHI) wherever possible and comply with HIPAA requirements [MF19] for data encryption at rest and in-transit, including appropriate Business Associate Agreements (BAA) for any cloud service providers [FKR+16]. For physical paper, HIPAA requirements mean that PHI must always be kept in a locked room or a container with restricted access.

Figure 1 shows a high level view of the user experience and information flow. Making appointments can be challenging, especially those with limited caregiver support. Because the appointment systems were set up in a hurry, many user interfaces were confusing and poorly designed. In the depicted pipeline, the person (or caregiver) telephones the 2-1-1 call center and the live operator collects demographic and health information, and coordinates any necessary travel arrangements, as needed. The demographic and health information is entered into the appointment system managed by the California Department of Public Health. The information is then downloaded to the clinic from the appointment system the day before the scheduled vaccination. Next, a forms packet is generated for every scheduled patient and consolidated into a PDF file that is then printed and handed to the volunteers at the clinic. The packet consolidates documents including consent forms, health forms, and CDC-provided vaccination cards.

When the patient arrives at the clinic, their forms are pulled and a volunteer reviews the questions while correcting any errors. Once the information is validated, the patient is directed to sign the appropriate forms. The crucially efficient part is that the patient and volunteer only have to *validate* previously collected information instead of filling out multiple forms with redundant information. This was crucial during peak demand so that most patients experienced less than a five minute delay between arrival and vaccine administration. While there was consideration of commercial services to do the electronic form filling and electronic signatures, they were discounted because these turned out to be too expensive and time-consuming to set up.

Different entities such as 2-1-1 and the State of California handle certain elements of the data pipeline, but strict HIPAA requirements are followed at each step. All clinic communications with the State appointment system were managed through a properly authenticated and encrypted system. The vaccine clinic utilized pre-existing, cloud-based HIPAA-compliant system, with corresponding BAAs. All sensitive data processing occurred on this system. The system, which is described at [HmLAKJU20], uses both python alone and in Jupyter notebooks.

Finally, the processed PDF forms were transferred using encryption to a server at the clinic site where an authorized operator printed them out. The paper forms were placed in the custody of a clinic volunteer until they were delivered to a back office for storage in a locked cabinet, pursuant to health department regulations.

Though all aspects of the pipeline faced challenges, the pre-population of forms turned out to be surprisingly difficult due to the lack of programmatic PDF tools that properly work with fillable forms. The remainder of the paper discusses the challenges and provides instructions on how to use Python to fill PDF forms for printing.

Programmatically Fill Forms

Programmatically filling in PDF forms can be a quick and accurate way to disseminate forms. Bits and pieces can be found throughout the Internet and places like Stack Overflow but no single source provides a complete answer. The *Medium* blog post by Vivsvaan Sharma [Sha20] is a good starting place. Another useful resource is the PDF 1.7 specification [pdf08]. Since the deployment of the vaccine clinic, the details of the form filling can be found at WestHealth's blog [Lu21]. The code is available on GitHub as described below.

The following imports are used in the examples given below.

```
import pdfrw
from pdfrw.objects.pdfstring import PdfString
from pdfrw.objects.pdfstring import BasePdfName
from pdfrw import PdfDict, PdfObject
```

Finding Your Way Around PDFrw and Fillable Forms

Several examples of basic form filling code can be found on the Internet, including the above-mentioned *Medium* blog post. The following is a typical snippet which was taken largely from the blog post.

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

The type of `annotation['/T']` is `pdfString`. While some sources use `[1:-1]` to extract the string from `pdfString`, the `to_unicode` method is the proper way to extract the string. According to the PDF 1.7 specification § 12.5.6.19, all fillable forms use widget annotation. The check for `annotation['/SubType']` filters the annotations to only widget annotations.

To set the value `value`, a `PDFString` needs to be created by encoding `value` with the `encode` method. The encoded `PDFString` is then used to update the `annotation` as shown in the following code snippet.

```
annotation.update(PdfDict(V=PdfString.encode(value)))
```

This converts `value` into a `PdfString` and updates the `annotation`, creating a value for `annotation['/V']`.

In addition, at the top level of the `PdfReader` object `pdf`, the `NeedAppearances` property in the interactive form dictionary, `AcroForm` (See § 12.7.2) needs to be set, without this, the fields are updated but will not necessarily display. To remedy this, the following code snippet can be used.

```
pdf.Root.AcroForm.update(PdfDict(
    NeedAppearances=PdfObject('true')))
```

Multiple Fields with Same Name

Combining the code snippets provides a simple method for filling in text fields, except if there are multiple instances of the same field. To refer back to the clinic example, each patient's form packet comprised multiple forms each with the `Name` field. Some forms even had the `Name` appear twice such as in a demographic section and then in a `Print Name` field next to a signature line. If the code above on such a form were run, the `Name` field will not show up.

Whenever the multiple fields occur with the same name, the situation is more complicated. One way to deal with this is to simply rename the fields to be different such as `Name-1` and `Name-2`, which is fine if the sole use of the form is for automated form filling. This would require access to a form authoring tool. If the form is also to be used for manual filling, this would require the user to enter the `Name` multiple times.

When fields appear multiple times, the widget annotation does not have the `/T` field but has a `/Parent` field. As it turns out this `/Parent` contains the field name `/T` as well as the default value `/V`. Each `/Parent` has one `/Kids` for each occurrence of the field. To modify the code to handle repeated occurrences of a field, the following lines can be inserted:

```
if not annotation['/T']:
    annotation=annotation['/Parent']
```

These lines allow the inspection and modifications of annotations that appear more than once. With this modification, the result of the inspection code yields:

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if not annotation['/T']:
                annotation=annotation['/Parent']
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

With this code in the above example, `Name` would be printed multiple times, once for each instance, but each instance points to the same `/Parent`. With this modification, the form filler actually fills the `/Parent` value multiple times, but this has no impact since it is overwriting the default value with the same value.

Checkboxes

In accordance to §12.7.4.2.3, the checkbox state can be set as follows:

```
def checkbox(annotation, value):
    if value:
        val_str = BasePdfName('/Yes')
    else:
        val_str = BasePdfName('/Off')
    annotation.update(PdfDict(V=val_str))
```

This could work if the export value of the checkbox is `Yes`, which is the default, but not when the export value is something else. The easiest solution is to edit the form to ensure that the export value of the checkbox is `Yes` and the default state of the box is unchecked. The recommendation in the specification is that it be set to `Yes`. In the event tools to make this change are not available, the `/V` and `/AS` fields should be set to the export value not `Yes`. The export value can be inspected by examining the appearance dictionary `/AP` and specifically at the `/N` field. Each annotation has up to three appearances in its appearance dictionary: `/N`, `/R` and `/D`, standing for *normal*, *rollover*, and *down* (§12.5.5). The latter two have to do with appearance in interacting with the mouse. The normal appearance has to do with how the form is printed.

There may be circumstances where the form has checkboxes whose default state is checked. In that case, in order to uncheck a box, the best practice is to delete the `/V` as well as the `/AS` field from the dictionary.

According to the PDF specification for checkboxes, the appearance stream `/AS` should be set to the same value as `/V`. Failure to do so may mean that the checkboxes do not appear.

More Complex Forms

For the purpose of the vaccine clinic application, the filling of text fields and checkboxes were all that were needed. However, for completeness, other form field types were studied and solutions are given below.

Radio Buttons

Radio buttons are by far the most complex of the form entry types. Each widget links to `/Kids` which represent the other buttons in the radio group. Each widget in a radio group will link to the same 'kids'. Much like the 'parents' for the repeated forms fields with the same name, each kid need only be updated once, but the same update can be used multiple times if it simplifies the code.

In a nutshell, the value `/V` of each widget in a radio group needs to be set to the export value of the button selected. In each kid, the appearance stream `/AS` should be set to `/Off` except for the kid corresponding to the export value. In order to identify the kid with its corresponding export value, the `/N` field of the appearance dictionary `/AP` needs to be examined just as was done with the checkboxes.

The resulting code could look like the following:

```
def radio_button(annotation, value):
    for each in annotation['/Kids']:
        # determine the export value of each kid
        keys = each['/AP']['/N'].keys()
        keys.remove('/Off')
        export = keys[0]

        if f'/{value}' == export:
            val_str = BasePdfName(f'/{value}')
        else:
            val_str = BasePdfName(f'/Off')
        each.update(PdfDict(AS=val_str))

    annotation.update(PdfDict(
        V=BasePdfName(f'/{value}')))
```

Combo Boxes and Lists

Both combo boxes and lists are forms of the form type *choice*. The combo boxes resemble drop-down menus and lists are similar to list pickers in HTML. Functionally, they are very similar in form filling. The value `/V` and appearance stream `/AS` need to be set to their exported values. The `/Op` field yields a list of lists associating the exported value with the value that appears in the widget.

To set the combo box, the value needs to be set to the export value.

```
def combobox(annotation, value):
    export=None
    for each in annotation['/Opt']:
        if each[1].to_unicode()==value:
            export = each[0].to_unicode()
    if export is None:
        err = f"Export Value: ""{value} Not Found"
        raise KeyError(err)
    pdfstr = PdfString.encode(export)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

Lists are structurally very similar. The list of exported values can be found in the `/Opt` field. The main difference is that lists based on their configuration can take multiple values. Multiple values can be set with `PDFrw` by setting `/V` and `/AS` to a list of `PdfStrings`. The code presented here uses two separate helpers, but because of the similarity in structure between list boxes and combo boxes, they could be combined into one function.

```
def listbox(annotation, values):
    pdfstrs=[]
    for value in values:
        export=None
        for each in annotation['/Opt']:
            if each[1].to_unicode()==value:
                export = each[0].to_unicode()
        if export is None:
            err = f"Export Value: {value} Not Found"
            raise KeyError(err)
        pdfstrs.append(PdfString.encode(export))
    annotation.update(PdfDict(V=pdfstrs, AS=pdfstrs))
```

Determining Form Field Types Programmatically

While PDF authoring tools or visual inspection can identify each form's type, the type can be determined programmatically as well. It is important to understand that fillable forms fall into four form types, button (push button, checkboxes and radio buttons), text, choice (combo box and list box), and signature. They correspond to following values of the `/FT` form type field of a given annotation, `/Btn`, `/Tx`, `/Ch` and `/Sig`, respectively. Since signature filling is not supported and the push button is a widget which can cause an action but is not fillable, those corresponding types are omitted from consideration.

To distinguish the types of buttons and choices, the form flags `/Ff` field is examined. For radio buttons, the 16th bit is set. For combo box the 18th bit is set. Please note that `annotation['/Ff']` returns a `PdfObject` when returned and must be coerced into an `int` for bit testing.

```
def field_type(annotation):
    ft = annotation['/FT']
    ff = annotation['/Ff']

    if ft == '/Tx':
        return 'text'
    if ft == '/Ch':
        if ff and int(ff) & 1 << 17:  # test 18th bit
            return 'combo'
        else:
            return 'list'
    if ft == '/Btn':
        if ff and int(ff) & 1 << 15:  # test 16th bit
            return 'radio'
        else:
            return 'checkbox'
```

For completeness, the following `text_form` filler helper is included.

```
def text_form(annotation, value):
    pdfstr = PdfString.encode(value)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

This completes the building blocks to an automatic form filler.

Consolidating Multiple Filled Forms

There are two problems with consolidating multiple filled forms. The first problem is that when two PDF files are merged, fields with matching names are associated with each other. For instance, if John Doe were entered in one form's name field and Jane Doe in the second. After combining the two forms John Doe will override the second form's name field and John Doe would appear in both forms. The second problem is that most simple command line or programmatic methods of combining two or more PDF files lose form data. One solution is to "flatten" each PDF file. This is equivalent to printing the file to PDF. In effect, this bakes in the filled form values and does not permit the editing the fields. Going even further, one could render the PDFs as images if the only requirement is that the combined files be printable. However, tools like `ghostscript`, `imagemagick`, and PDFUnite don't do a good job of preserving form data when rendering PDF files.

Form Field Name Collisions

Combining multiple filled PDF files was an issue for the vaccine clinic because the same form was filled out for multiple patients. The alternative of printing hundreds of individual forms was infeasible. To combine a batch of PDF forms, all form field names must be different. Thankfully, the solution is quite simple, in the process of filling out the form using the code above, rename (set) the value of `/T`.

```
def form_filler(in_path, data, out_path, suffix):
    pdf = pdfrw.PdfReader(in_path)
    for page in pdf.pages:
        annotations = page['/Annots']
        if annotations is None:
            continue

        for annotation in annotations:
            if annotation['/SubType'] == '/Widget':
                key = annotation['/T'].to_unicode()
                if key in data:
                    pdfstr = PdfString.encode(data[key])
                    new_key = key + suffix
                    annotation.update(
                        PdfDict(V=pdfstr, T=new_key))
        pdf.Root.AcroForm.update(PdfDict(
             NeedAppearances=PdfObject('true')))
        pdfrw.PdfWriter().write(out_path, pdf)
```

Only a unique suffix needs to be supplied to each form. The suffix can be as simple as a sequential number.

Combining the Files

Solutions for combining PDF files with `PDFrw` can be found on the Internet. The following recipe is typical:

```
writer = PdfWriter()
for fname in files:
    r = PdfReader(fname)
    writer.addpages(r.pages)
writer.write("output.pdf")
```

While the form data still exists in the output file, the rendering information is lost and won't show when displayed or printed. The problem comes from the fact that the written PDF does not have an interactive form dictionary (see §12.7.2 of the PDF 1.7 specification). In particular, the interactive forms dictionary contains the boolean `NeedAppearances` which needs to be set for fields to be shown. If the forms being combined have different interactive form dictionaries, they need to be merged. In this application where the source forms are identical among the various copies, any `AcroForm` dictionary can be used.

After obtaining the dictionary from `pdf.Root.AcroForm` (assuming the `PdfReader` object is stored in `pdf`), it is not clear how to add it to the `PdfWriter` object. The clue comes from a simple recipe for copying a pdf file.

```
pdf = PdfReader(in_file)
PdfWriter().write(out_file, pdf)
```

Examination of the underlying source code shows the second parameter `pdf` to be set to the attribute `trailer` of the `PdfWriter` object. Assuming `acro_form` contains the desired interactive form, the interactive form dictionary can be added to the output document by using `writer.trailer.Root.AcroForm = acro_form`.

Conclusion

A complete functional version of this PDF form filler is open source and can be found at WestHealth's GitHub repository https://github.com/WestHealth/pdf-form-filler. This process was able to produce large quantities of pre-populated forms for senior citizens seeking COVID-19 vaccinations relieving one of the bottlenecks that have plagued many other vaccine clinics.

References

[Col21] Annica Colbert. Seniors-only vaccination site. *KPBS News*, Feb 2021. URL: https://www.kpbs.org/podcasts/san-diego-news-now/2021/feb/11/seniors-only-vaccination-site/.

[FKR+16] Barbara L. Filkins, Ju Young Kim, Bruce Roberts, Winston Armstrong, Mark A. Miller, Michael L. Hultner, Anthony P. Castillo, Jean Christophe Ducom, Eric J. Topol, and Steven R. Steinhubl. Privacy and security in the era of digital health: What should translational researchers know and do about it? *American Journal of Translational Research*, 8(3):1560–1580, 2016. Publisher Copyright: © 2016, E-Century Publishing Corporation. All rights reserved.

[HmLAKJU20] Haw-minn Lu, Adrian Kwong, and José Unpingco. Securing Your Collaborative Jupyter Notebooks in the Cloud using Container and Load Balancing Services. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 2 – 10, 2020. doi:10.25080/Majora-342d178e-001.

[Let21] U-T Letters. Opinion: Vaccination frustration grows as seniors weigh limited options. *San Diego Union-Tribune*, Jan 2021. URL: https://www.sandiegouniontribune.com/opinion/story/2021-01-22/vaccination-frustrations-grow-as-seniors-search-for-appointments.

[Lit21] Joe Little. For san diego seniors, making vaccination appointments is as easy as calling 211. *7 San Diego News*, Feb 2021. URL: https://www.nbcsandiego.com/news/local/for-san-diego-seniors-making-vaccination-appointments-is-as-easy-as-calling-211/2531346/.

[Lu21] Haw-minn Lu. Exploring fillable forms with pdfrw, Mar 2021. URL: https://westhealth.github.io/exploring-fillable-forms-with-pdfrw.html.

[MF19] Wilnellys Moore and Sarah Frye. Review of hipaa, part 1: History, protected health information, and privacy and security rules. *Journal of Nuclear Medicine Technology*, 47(4):269–272, 2019. URL: https://tech.snmjournals.org/content/47/4/269, arXiv:https://tech.snmjournals.org/content/47/4/269.full.pdf, doi:10.2967/jnmt.119.227819.

[pdf08] *Document Management - Portable Document Format - Part 1: PDF 1.7*. Adobe Systems Incorporated, 2008.

[pod21] Pop up vaccination point of distribution (pod) for seniors: A how to guide, Apr 2021. URL: https://www.westhealth.org/resource/vaccine-pod-for-seniors/.

[pre21] New covid-19 vaccine site for vulnerable seniors at west pace in san marcos. *West Health Press Releases*, Feb 2021. URL: https://www.westhealth.org/press-release/new-covid-19-vaccine-site-for-vulnerable-seniors-at-west-pace-in-san-marcos/.

[Sha20] Vivsvaan Sharma. Filling editable pdf in python, Aug 2020. URL: https://medium.com/@vivsvaan/filling-editable-pdf-in-python-76712c3ce99.

Using Python for Analysis and Verification of Mixed-mode Signal Chains

Mark Thoren‡, Cristina Suteu‡*

Abstract—Any application involving sensitive measurements of the physical world starts with accurate, precise, and low-noise signal chain. Modern, highly integrated data acquisition devices can often be directly connected to sensor outputs, performing analog signal conditioning, digitization, and digital filtering on a single silicon device, greatly simplifying system electronics. However, a complete understanding of the signal chain's noise sources and noise limiting filters is still required to extract maximum performance from and debug these modern devices.

Introduction

Mixed-mode signal chains are everywhere. Simply put, any system that transforms a real-world signal to an electrical representation, which is then digitized can be classified as a mixed-mode signal chain. At every point along the chain the signal is degraded in various ways that can usually be characterized either as some form of distortion or additive noise. Once in the digital domain, the processing of the digitized data is not perfect either, but at least it is, for all practical purposes, immune to many of the offenders that affect analog signals - component tolerances, temperature drift, interference from adjacent signals or supply voltage variations.

As the industry continues to push the physical limits, one thing that can be stated with certainty is this: there is always room for improvement in analog and mixed signal components for instrumentation. If an Analog to Digital Converter (ADC) or a Digital to Analog Converter (DAC) appears on the market that advances the state of the art in speed, noise, power, accuracy, or price, industry will happily apply it to existing problems, then ask for more improvement. However, in order to achieve the best acquisition system for your application, it is fundamental to be aware of the components' limitations and choose these accordingly.

This tutorial is in extension of Converter Connectivity Tutorial[1] and associated code and simulation files[2]. A representative signal chain will be analyzed and tested, focusing on noise. Individual signal chain elements will first be modelled with the help of Python / SciPy[3] and LTspice[4], then verified using Python to drive low-cost instrumentation and evaluation boards via the Linux Industrial Input Output (IIO) framework. While primarily for the education space, these instruments have adequate performance for many industrial applications. Furthermore, these techniques can easily be adapted to other bench-top instruments.

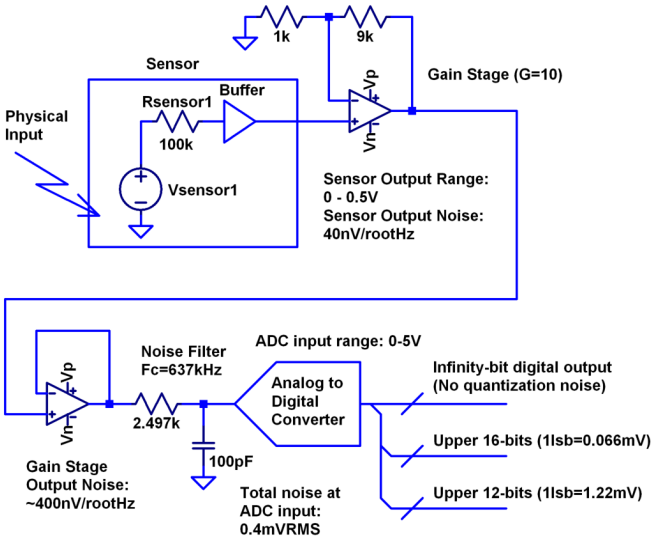

Fig. 1: *In a mixed-mode signal chain, some physical phenomenon such as temperature, light intensity, pH, force, or torque is converted to an electrical parameter (resistance, current, or directly to voltage). This signal is then amplified, low-pass filtered, and digitized by an ADC, which may include internal digital filtering.*

A Generic Mixed Signal Chain

Figure 1 shows a generic signal chain typical of a precision instrumentation application, with a physical input and digital output. There are numerous background references on analog to digital converters available[5], and most readers will have a sense that an analog to digital converter samples an input signal at some point in time (or measures the average of a signal over some observation time), and produces a numerical representation of that signal - most often as a binary number with some value between zero and $2^N - 1$ where N is the number of bits in the output word.

ADC Noise Sources

While there are several noise sources in Figure 1, one that is often either ignored, or over-emphasized, is the number of bits in the ADC's digital output. Historically, an ADC's "number of bits" was considered the ultimate figure of merit, where a 16-bit converter was 4 times better than a 14-bit converter[6]. But in the case of

‡ *Analog Devices, Inc.*
* *Corresponding author: cristina.suteu@analog.com*

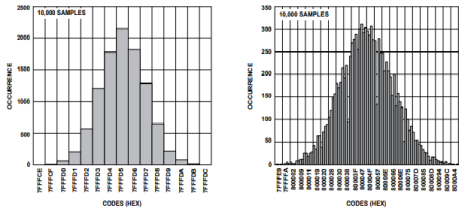

Fig. 2: *At a PGA gain of one (left), 13 codes are represented in the AD7124 output noise, and the standard deviation is about 2.5 codes. While quantization is visible, thermal noise is more significant. At a PGA gain of 128 (right), 187 codes are represented, quantization noise is insignificant. Truncating one or two least-significant bits (doubling or quadrupling quantization noise) would not result in a loss of information.*

modern, high-resolution converters, the "number of bits" can be safely ignored. Note a general principle of signal chain design:

"The input noise of one stage should be somewhat lower than the output noise of the preceding stage."

As with any signal chain, one noise source within an ADC often dominates. Thus, if a noiseless signal applied to an N-bit ADC:

- results in either a single output code, or two adjacent output codes, then **quantization noise dominates**. The Signal to Noise Ratio can be no greater than (6.02 N + 1.76) dB[7].
- results in a gaussian distribution of "many" output codes, then **thermal noise source dominates**. The Signal to Noise Ratio is no greater than:
  $20\log(V_{in}(p-p)/(\sigma/\sqrt{8}))$, where:
  $V_{in}(p-p)$ is the full-scale input signal
  σ is the standard deviation of the output codes in units of voltage.

Very high resolution converters, such as the AD7124-8 that will be used as an example shortly, are rarely limited by quantization noise; thermal noise dominates in all of the gain / bandwidth settings, and a shorted input will always produce a fairly Gaussian distribution of output codes. Figure 2 , from Ref.[8] shows the grounded-input histogram of the AD712482, 24-bit sigma-delta ADC, with the internal Programmable Gain Amplifier (PGA) set to 1 and 128, respectively.

Modeling and Measuring ADC noise

Modeling the noise of a thermal-noise limited ADC's is straightforward. If the noise is "well behaved" (Gaussian, as it is in Figure 2) and constant across the ADC's input span, the ADC's time-domain noise can be modelled using NumPy's[9] `random.normal` function , then verified by taking the standard deviation, as seen in the Model Gaussian Noise code block.

```
# Model Gaussian Noise
# See AD7124 datasheet for noise levels per mode
import numpy as np
offset = 0.000
rmsnoise = 0.42e-6  # AD7124 noise
noise = np.random.normal(loc=offset, scale=rmsnoise,
                         size=1024)
measured_noise = np.std(noise)
print("Measured Noise: ", measured_noise)
```

Figure 3 shows the general setup for testing ADC noise and filter response[1].

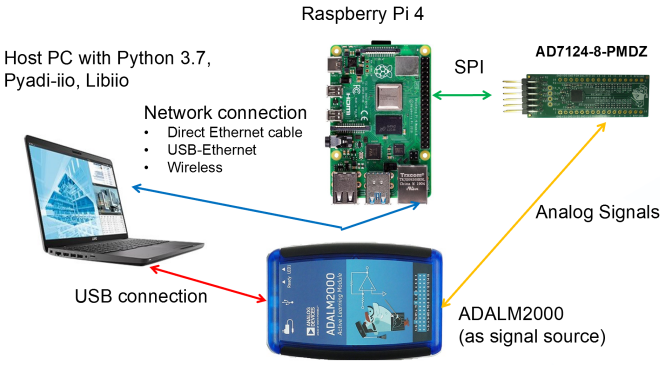

Fig. 3: *The ADALM2000 is a multifunction USB test instrument with two general-purpose analog inputs and two outputs, with sample rates of 100Msps and 150Msps, respectively. It can be used as a simple signal source for measuring ADC noise bandwidth and filter response. A Raspberry Pi 4 running a kernel with AD7124 device driver support acts as a simple bridge between the AD7124 and a host computer.*

The AD7124 device driver falls under the industry-standard IIO framework, which has a well-established software API (including Python bindings). Application code can run locally (on the Raspberry Pi) or on a remote machine via network, serial, or USB connection. Furthermore, the pyadi-iio[10] abstraction layer takes care of much of the boilerplate setup required for interfacing with IIO devices, greatly simplifying the software interface. The AD7124-8 Basic Data Capture code block illustrates how to open a connection to the AD7124-8, configure it, capture a block of data, then close the connection.

```
 # AD7124-8 Basic Data Capture

import adi  # pyadi-iio library
# Connect to AD7124-8 via Raspberry Pi
my_ad7124 = adi.ad7124(uri="ip:analog.local")
ad_channel = 0  # Set channel
# Set PGA gain
my_ad7124.channel[ad_channel].scale = 0.0002983
my_ad7124.sample_rate = 128  # Set sample rate
# Read a single "raw" value
v0 = my_ad7124.channel[ad_channel].raw
# Buffered data capture
my_ad7124.rx_output_type = "SI"  # Report in volts
# Only one buffered channel supported for now
my_ad7124.rx_enabled_channels = [ad_channel]
my_ad7124.rx_buffer_size = 1024
my_ad7124._ctx.set_timeout(100000)  # Slow
data = my_ad7124.rx()  # Fetch buffer of samples

print("A single raw reading: ", v0)
print("A few buffered readings: ", data[:16])
del my_ad7124  # Clean up
```

With communication to the AD7124-8 established, an extremely simple, yet extremely useful test can be performed: measuring input noise directly. Simply shorting the input to an ADC and looking at the resulting distribution of ADC codes is a valuable step in characterizing a signal chain design. The AD7124 input mode is set to unipolar, so only positive values are valid; the test circuit shown in Figure 4 ensures that the input is always positive.

Figure 5 shows two, 1024-point measurements. The lower (blue) trace was taken immediately after initially applying power.

The "wandering" can be due to a number of factors - the internal reference warming up, the external resistors warming up (and hence drifting), or parasitic thermocouples, where slightly dissimilar metals will produce a voltage in the presence of

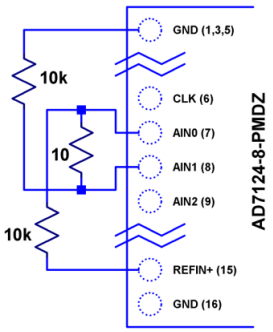

Fig. 4: *A resistor divider is used to generate a 1.25mV bias across the AD7124-8's input, overcoming the 15µV maximum offset voltage and ensuring that ADC readings are always positive.*

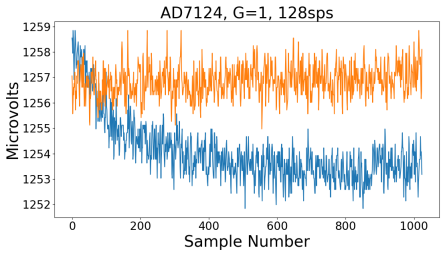

Fig. 5: *Two AD7124-8 data captures are taken with a 1.25mV bias applied. The lower trace shows initial drift after power-up as the circuit warms up. The upper trace shows stable readings after a half-hour warmup time.*

thermal gradients. Measured noise after warmup is approximately 565nVRMS - on par with the datasheet noise specification.

Expressing ADC Noise as a Density

The general principle of analog signal chain design (that the input noise of one stage should be somewhat lower than the output noise of the preceding stage) is an easy calculation if all elements include noise density specifications, as most well-specified sensors, and nearly all amplifiers do.

Unlike amplifiers and sensors, ADC datasheets typically do not include a noise density specification. Expressing the ADC's noise as a density allows it to be directly compared to the noise at the output of the last element in the analog signal chain, which may be an ADC driver stage, a gain stage, or the sensor itself.

An ADC's internal noise will necessarily appear somewhere between DC and half the sample rate. Ideally this noise is flat, or at least predictably shaped. In fact, since the ADC's total noise is spread out across a known bandwidth, it can be converted to a noise density that can be directly compared to other elements in the signal chain. Precision converters typically have total noise given directly, in volts RMS: $e_{RMS} = \sigma$, where e_{RMS} is the total RMS noise, calculated from the standard deviation of a grounded-input histogram of codes.

Higher speed converters that are tested and characterized with sinusoidal signals will typically have a signal to noise (SNR) specification. If provided, the total RMS noise can be calculated as:

$$e_{RMS} = \frac{ADCp-p}{\sqrt{8} * 10^{\frac{SNR}{20}}}$$

Where $ADCp-p$ is the peak-to-peak input range of the ADC.

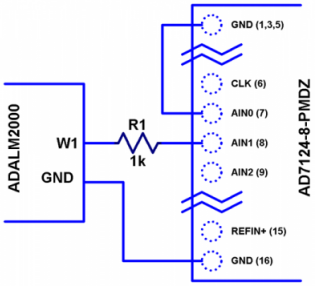

Fig. 6: *An ADALM2000 waveform generator is used to generate a range of sinewave frequencies, allowing the AD7124-8's filer response to be measured directly. While the script sets the sinewave amplitude and offset to a safe level, a 1k resistor protects the AD7124-8 in the event of a malfunction. (The ADALM2000 output voltage range is -5V to +5V, while the AD7124-8 absolute maximum limits are -0.3V and +3.6V.)*

The equivalent noise density can then be calculated:

$$e_n = \frac{e_{RMS}}{\sqrt{\frac{fs}{2}}}$$

Where fs is the ADC sample rate in samples/second.

The total noise from Figure 5 after warmup was 565nV at a data rate of 128sps. The noise density is approximately:

$$565nV/\sqrt{64Hz} = 70nV/\sqrt{Hz}$$

The ADC can now be directly included in the signal chain noise analysis, and leads to a guideline for optimizing the signal chain's gain:

Increase the gain just to the point where the noise density of the last stage before the ADC is a bit higher than that of the ADC, then stop. Don't bother increasing the signal chain gain any more - you're just amplifying noise and decreasing the allowable range of inputs.

This runs counter to the conventional wisdom of "filling" the ADC's input range. There may be benefit to using more of an ADC's input range if there are steps or discontinuities in the ADC's transfer function, but for "well behaved" ADCs (most sigma delta ADCs and modern, high-resolution Successive Approximation Register (SAR) ADCs), optimizing by noise is the preferred approach.

Measuring ADC filter response

The AD7124-8 is a sigma-delta ADC, in which a modulator produces a high sample rate, but noisy (low resolution), representation of the analog input. This noisy data is then filtered by an internal digital filter, producing a lower rate, lower noise output. The type of filter varies from ADC to ADC, depending on the intended end application. The AD7124-8 is general-purpose, targeted at precision applications. As such, the digital filter response and output data rate are highly configurable. While the filter response is well-defined in the datasheet, there are occasions when one may want to measure the impact of the filter on a given signal. The AD7124-8 Filter Response code block measures the filter response by applying sinewaves to the ADC input and analyzing the output. This method can be easily adapted to measuring other waveforms - wavelets, simulated physical events. The ADALM2000 is connected to the AD7124-8 circuit as shown in Figure 6.

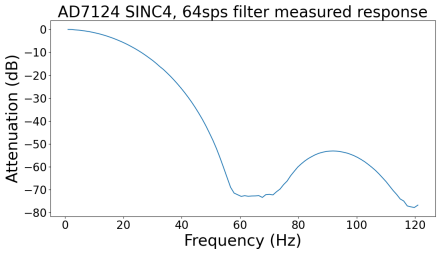

Fig. 7: *A measurement of the AD7124 filter response in 64sps, SINC4 mode shows the filter's passband, first lobe, and first two nulls.*

The AD7124-8 Filter Response code block will set the ADALM2000's waveform generator to generate a sinewave at 10Hz, capture 1024 data points, calculate the RMS value, then append the result to a list. (The `send_sinewave` and `capture_data` are utility functions that send a sinewave to the ADALM2000 and receive a block of data from the AD7124, respectively[2].) It will then step through frequencies up to 250Hz, then plot the result as shown in Figure 7.

```
# AD7124-8 Filter Response
import numpy as np
import matplotlib.pyplot as plt
resp = []
freqs = np.linspace(1, 121, 100, endpoint=True)
for freq in freqs:
    print("testing ", freq, " Hz")
    send_sinewave(my_siggen, freq)   # Set frequency
    time.sleep(5.0)                  # Let settle
    data = capture_data(my_ad7124)   # Grab data
    resp.append(np.std(data))   # Take RMS value
    if plt_time_domain:
        plt.plot(data)
        plt.show()
    capture_data(my_ad7124)   # Flush
# Plot log magnitude of response.
response_dB = 20.0 * np.log10(resp/0.5*np.sqrt(2))
print("\n Response [dB] \n")
print(response_dB)
plt.figure(2)
plt.plot(freqs, response_dB)
plt.title('AD7124 filter response')
plt.ylabel('attenuation')
plt.xlabel("frequency")
plt.show()
```

While measuring high attenuation values requires a quieter and lower distortion signal generator, the response of the first few major "lobes" is apparent with this setup.

Modeling ADC filters

The ability to measure an ADC's filter response is a practical tool for bench verification. However, in order to fully simulate a signal chain, a model of the filter is needed. This isn't explicitly provided for many converters (including the AD7124-8), but a workable model can be reverse engineered from the information provided in the datasheet.

Note that what follows is only a model of the AD7124-8 filters, it is not a bit-accurate representation. Refer to the AD7124-8 datasheet for all guaranteed parameters.

The AD7124's filters all have frequency responses that are combinations of various SINC functions (with a frequency response proportional to $(sinf/f)^N$) . These filters are fairly easy to construct, and to reverse-engineer when nulls are known.

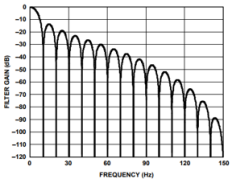

Fig. 8: *The AD7124-8 10Hz notch filter has a SINC1 magnitude response; the filter's impulse response is simply an unweighted (rectangular) average of samples over a 100ms time interval.*

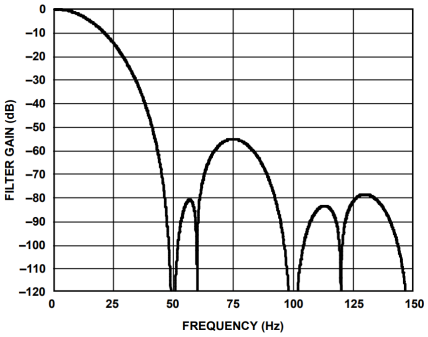

Fig. 9: *The AD7124-8 50/60Hz rejection filter response is the combination of a 50Hz, SINC3 filter and a 60Hz, SINC1 filter.*

Figure 8 from Ref.[8] shows the AD7124-8's 10Hz notch filters. Various combinations of higher-order SINC3 and SINC4 filters are also available.

The simultaneous 50Hz/60Hz rejection filter shown in Figure 9, from Ref.[8] is a nontrivial example. This filter is intended to strongly reject noise from A.C. power lines, which is either 50Hz (as in Europe) or 60Hz (as in the United States).

Higher order SINC filters can be generated by convolving SINC1 filters. For example, convolving two SINC1 filters (with a rectangular impulse response in time) will result in a triangular impulse response, and a corresponding SINC2 frequency response. The AD7124 Filters code block generates a SINC3 filter with a null at 50Hz, then adds a fourth filter with a null at 60Hz.

```
# AD7124 Filters
import numpy as np
f0 = 19200
# Calculate SINC1 oversample ratios for 50, 60Hz
osr50 = int(f0/50)   # 384
osr60 = int(f0/60)   # 320

# Create "boxcar" SINC1 filters
sinc1_50 = np.ones(osr50)
sinc1_60 = np.ones(osr60)

# Calculate higher order filters
sinc2_50 = np.convolve(sinc1_50, sinc1_50)
sinc3_50 = np.convolve(sinc2_50, sinc1_50)
sinc4_50 = np.convolve(sinc2_50, sinc2_50)

# Here's the SINC4-ish filter from datasheet
# Figure 91, with three zeros at 50Hz, one at 60Hz.
filt_50_60_rej = np.convolve(sinc3_50, sinc1_60)
```

The resulting impulse (time domain) shapes of the filters are shown in Figure 10. Filter coefficient (tap) values are normalized for unity (0dB) gain at zero frequency.

And finally, the frequency response can be calculated using NumPy's freqz function, as seen in the AD7124 Frequency Response code block. The response is shown in Figure 11.

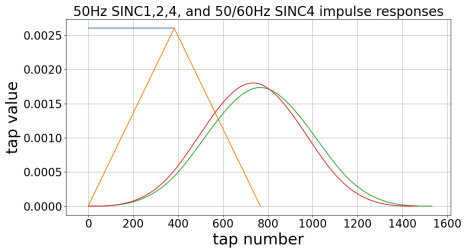

Fig. 10: *Repeatedly convolving rectangular impulse responses produces triangular, then "Gaussian-like" impulse responses.*

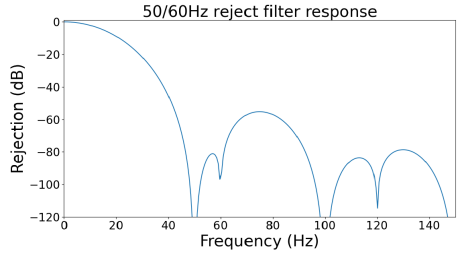

Fig. 11: *Convolving a SINC3, 50Hz notch filter with a SINC1, 60Hz filter produces a composite response that strongly rejects both 50Hz and 60Hz.*

```
# AD7124 Frequency Response
import numpy as np
from scipy import signal
f0 = 19200
w, h = signal.freqz(filt_50_60_rej, 1, worN=16385,
                    whole = False, fs = f0)
freqs = w * f0/(2.0*np.pi)
hmax = abs(max(h))  # Normalize to unity
response_dB = 20.0 * np.log10(abs(h)/hmax)
```

Resistance is Futile: A Fundamental Sensor Limitation

All sensors, no matter how perfect, have some maximum input value (and a corresponding maximum output - which may be a voltage, current, resistance, or even dial position) and a finite noise floor - "wiggles" at the output that exist even if the input is perfectly still. At some point, a sensor with an electrical output will include an element with a finite resistance (or more generally, impedance) represented by Rsensor in Figure 12. This represents one fundamental noise limit that cannot be improved upon - this resistance will produce, at a minimum:

$e_n(RMS) = \sqrt{4 * K * T * Rsensor * (F2 - F1)}$ Volts of noise, where:

$e_n(RMS)$ is the total noise

K is Boltzmann's constant (1.38e-23 J/K)

T is the resistor's absolute temperature (Kelvin)

F2 and F1 are the upper and lower limits of the frequency band of interest.

Normalizing the bandwidth to 1Hz expresses the noise density, in $\frac{V}{\sqrt{Hz}}$.

A sensor's datasheet may specify a low output impedance (often close to zero Ohms), but this is likely a buffer stage - which eases interfacing to downstream circuits, but does not eliminate noise due to impedances earlier in the signal chain.

There are numerous other sensor limitations - mechanical, chemical, optical, each with their own theoretical limits and whose

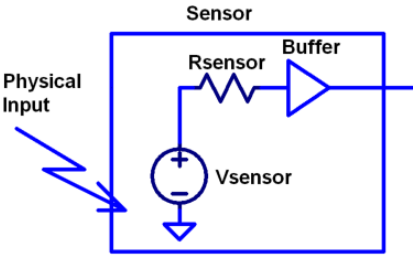

Fig. 12: *Sensors often include an internal buffer to simplify connection to downstream circuits. While the output impedance is low (often approaching zero Ohms), noise from high impedance sensing elements is buffered along with the signal.*

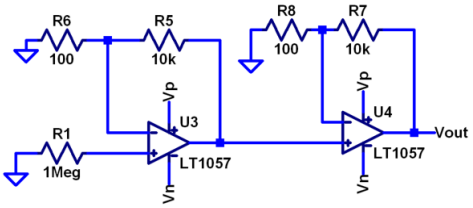

Fig. 13: *A 1M resistor serves as a predictable noise source, which is then amplified to a usable level by a low-noise operational amplifier.*

effects can be modelled and compensated for later. But noise is the one imperfection that cannot.

A Laboratory Noise Source

A calibrated noise generator functions as a "world's worst sensor", that emulates the noise of a sensor without actually sensing anything. Such a generator allows a signal chain's response to noise to be measured directly. The circuit shown in Figure 13 uses a 1M resistor as a 127nV/$\sqrt{Hz}$ (at room temperature) noise source with "okay accuracy" and bandwidth. While the accuracy is only "okay", this method has advantages:

- It is based on first principles, so in a sense can act as an uncalibrated standard.
- It is truly random, with no repeating patterns.

The OP482 is an ultralow bias current amplifier with correspondingly low current noise, and a voltage noise low enough that the noise due to a 1M input impedance is dominant. Configured with a gain of 2121, the output noise is 269 µV/$\sqrt{Hz}$.

The noise source was verified with an ADALM2000 USB instrument, using the Scopy[11] GUI's spectrum analyzer, shown in Figure 14.

Under the analyzer settings shown, the ADALM2000 noise floor is 40µV/$\sqrt{Hz}$, well below the 269 µV/$\sqrt{Hz}$ of the noise source.

While Scopy is useful for single, visual measurements, the functionality can be replicated easily with the SciPy periodogram function. Raw data is collected from an ADALM2000 using the libm2k[12] and Python bindings, minimally processed to remove DC content (that would otherwise "leak" into low frequency bins), and scaled to nV/$\sqrt{Hz}$. This method, shown in the Noise Source Measurement code block can be applied to any data acquisition module, so long as the sample rate is fixed and known, and data can be formatted as a vector of voltages.

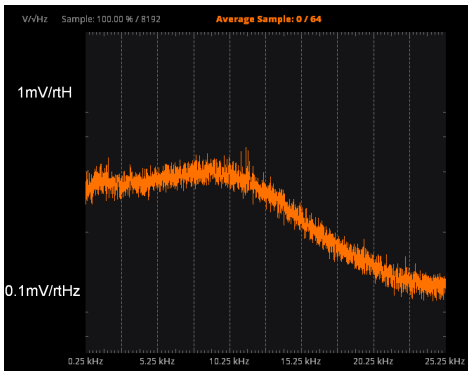

Fig. 14: *The output of the resistor-based laboratory noise generator has a usable bandwidth of approximately 10kHz.*

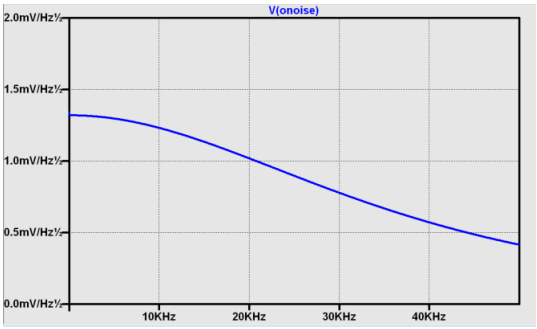

Fig. 15: *An LTspice simulation of the laboratory noise source shows approximately the same usable bandwidth as the measured circuit.*

```
# Noise Source Measurement
import numpy as np
navgs = 32  # Avg. 32 runs to smooth out data
ns = 2**16
vsd = np.zeros(ns//2+1)  # /2 for onesided
for i in range(navgs):
  ch1 = np.asarray(data[0])  # Extract ch 1 data
  ch1 -= np.average(ch1)  # Remove DC
  fs, psd = periodogram(ch1, 1000000,
                        window="blackman",
                        return_onesided=True)
  vsd += np.sqrt(psd)
vsd /= navgs
```

We are now armed with a known noise source and a method to measure said source, both of which can be used to validate signal chains.

Modeling Signal Chains in LTspice

LTspice is a freely available, general-purpose analog circuit simulator that can be applied to signal chain design. It can perform transient analysis, frequency-domain analysis (AC sweep), and noise analysis, the results of which can be exported and incorporated into mixed signal models using Python.

Figure 15 shows a noise simulation of the analog noise generator, with close agreement to experimental results. An op-amp with similar properties to the OP482 was used for the simulation.

Figure 15 circuit's noise is fairly trivial to model, given that it is constant for some bandwidth (in which a signal of interest would lie), above which it rolls off with approximately a first order lowpass response. Where this technique comes in handy is modeling non-flat noise floors, either due to higher order analog filtering, or active elements themselves. The classic example is the

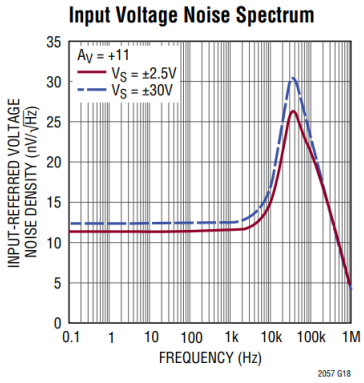

Fig. 16: *The LTC2057 noise density is flat at low frequencies, with a peak at 50kHz (half of the internal oscillator's 100kHz frequency).*

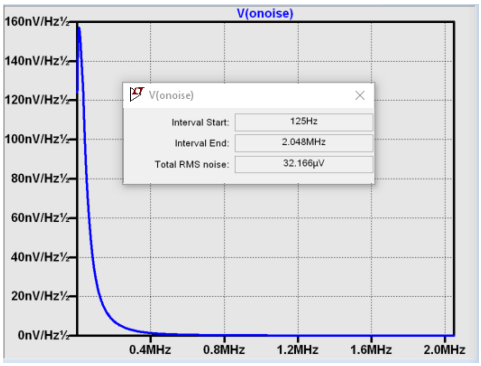

Fig. 17: *LTspice is used to simulate the output nosie of an LTC2057 in a noninverting gain of +10 configuration. LTspice provides simple tools for integrating noise, but results of any simulation can be exported and imported into Python for further analysis.*

"noise mountain" that often exists in autozero amplifiers such as the LTC2057, as seen in Figure 16 , from Ref.[13].

Importing LTspice noise data for frequency domain analysis in Python is a matter of setting up the simulation command such that exact frequencies in the analysis vector are simulated. In this case, the noise simulation is set up for a simulation with a maximum frequency of 2.048MHz and resolution of 62.5Hz , corresponding to the first Nyquist zone at a sample rate of 4.096 MSPS. Figure 17 shows the simulation of the LT2057 in a non-inverting gain of 10, simulation output, and exported data format.

In order to determine the impact of a given band of noise on a signal (signal to noise ratio) the noise is root-sum-square integrated across the bandwidth of interest. In LTspice, plotted parameters can be integrated by setting the plot limits, then control-clicking the parameter label. The total noise over the entire 2.048MHz simulation is 32µVRMS. A function to implement this operation in Python is shown in the Integrate Power Spectral Density code block.

```
def integrate_psd(psd, bw):
    import numpy as np
    int_psd_sqd = np.zeros(len(psd))
    integrated_psd = np.zeros(len(psd))
    int_psd_sqd[0] = psd[0]**2.0
    for i in range(1, len(psd)):
        int_psd_sqd[i] += int_psd_sqd[i-1]\
            + psd[i-1] ** 2
        integrated_psd[i] += int_psd_sqd[i]**0.5
    integrated_psd *= bw**0.5
    return integrated_psd
```

Reading in the exported noise data and passing to the integrate_psd function results in a total noise of 3.21951e-05, very close to LTspice's calculation.

Generating Test Noise

Expanding on the functionality of the purely analog noise generator above, it is very useful to be able to produce not only flat, but arbitrary noise profiles - flat "bands" of noise, "pink noise", "noise mountains" emulating peaking in some amplifiers. The Generate Time-series From Half-spectrum code block starts with a desired noise spectral density (which can be generated manually, or taken from an LTspice simulation), the sample rate of the time series, and produces a time series of voltage values that can be sent to a DAC.

```
def time_points_from_freq(freq, fs=1, density=False):
    import numpy as np
    N = len(freq)
    rnd_ph_pos = (np.ones(N-1, dtype=np.complex)*
                  np.exp(1j*np.random.uniform
                         (0.0, 2.0*np.pi, N-1)))
    rnd_ph_neg = np.flip(np.conjugate(rnd_ph_pos))
    rnd_ph_full = np.concatenate(([1], rnd_ph_pos, [1],
                                  rnd_ph_neg))
    r_s_full = np.concatenate((freq, np.roll
                               (np.flip(freq), 1)))
    r_spectrum_rnd_ph = r_s_full * rnd_ph_full
    r_time_full = np.fft.ifft(r_spectrum_rnd_ph)

    if (density is True):
        # Note that this N is "predivided" by 2
        r_time_full *= N*np.sqrt(fs/(N))
    return(np.real(r_time_full))
```

This function can be verified by controlling one ADALM2000 through a libm2k script, and verifying the noise profile with a second ADALM2000 and the spectrum analyzer in the Scopy GUI. The Push Noise Time-series to ADALM2000 code snippet generates four "bands" of 1mV/$\sqrt{Hz}$ noise on the ADALM2000 W2 output (with a sinewave on W1, for double-checking functionality.)

```
# Push Noise Time-series to ADALM2000
import numpy as np
n = 8192
# create some "bands" of  1mV/rootHz noise
bands = np.concatenate((np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16)))*1000e-6
bands[0] = 0.0  # Set DC content to zero
buffer2 = time_points_from_freq(bands, fs=75000,
                                density=True)
buffer = [buffer1, buffer2]
aout.setCyclic(True)
aout.push(buffer)
```

Figure 18 shows four bands of 1mV/$\sqrt{Hz}$ noise being generated by one ADALM2000. The input vector is 8192 points long at a sample rate of 75ksps, for a bandwidth of 9.1Hz per point. Each "band" is 512 points, or 4687Hz wide. The rolloff above ~20kHz is the SINC rolloff of the DAC. If the DAC is capable of a higher sample rate, the time series data can be upsampled and filtered by an interpolating filter[14].

This noise generator can be used in conjunction with the pure analog generator for verifying the rejection properties of a signal chain.

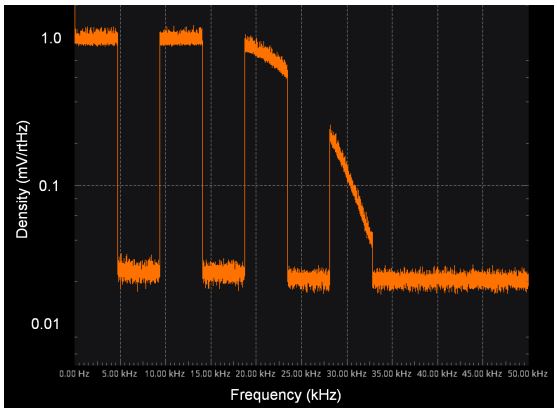

Fig. 18: *The Scopy spectrum analyzer is used to verify the arbitrary noise generator. Deep notches between noise bands expose the analyzer's noise floor, showing that an arbitrary noise profile can be accurately generated.*

Modeling and verifying ADC Noise Bandwidth

External noise sources and spurious tones above Fs/2 will fold back (alias) into the DC-Fs/2 region - and a converter may be sensitive to noise far beyond Fs/2 - the AD872A mentioned above has a sample rate of 10Msps, but an input bandwidth of 35MHz. While performance may not be the best at such high frequencies, this converter will happily digitize 7 Nyquist zones of noise and fold them back on top of your signal. This illustrates the importance of antialias filters for wideband ADCs. But converters for precision applications, which are typically sigma-delta (like the AD7124-8) or oversampling SAR architectures, in which the input bandwidth is limited by design.

It is often useful to think of the "equivalent noise bandwidth" (ENBW) of a filter, including an ADC's built-in filter. The ENBW is the bandwidth of a flat passband "brick wall" filter that lets through the same amount of noise as the non-flat filter. A common example is the ENBW of a first-order R-C filter, which is:

$$ENBW = fc * \pi/2$$

Where fc is the cutoff frequency of the filter. If broadband noise, from "DC to daylight", is applied to the inputs of both a 1KHz, first-order lowpass filter and 1.57kHz brickwall lowpass filter, the total noise power at the outputs will be the same.

The ENBW Example code block accepts a filter magnitude response, and returns the effective noise bandwidth. A single-pole filter's magnitude response is calculated, and used to verify the ENBW = $fc * pi/2$ relationship.

```
import numpy as np
def arb_enbw(fresp, bw):
    int_frsp_sqd = np.zeros(len(fresp))
    int_frsp_sqd[0] = fresp[0]**2.0
    for i in range(1, len(fresp)):
        int_frsp_sqd[i] += (int_frsp_sqd[i-1] +
                            fresp[i-1] ** 2)
    return int_frsp_sqd[len(int_frsp_sqd)-1]*bw

fc = 1  # Hz
bw_per_point = 200/65536  # Hz/record length
frst_ord = np.ndarray(65536, dtype=float)
# Magnitude = 1/SQRT(1 + (f/fc)^2))
for i in range(65536):
    frst_ord[i] = (1.0 /
                   (1.0 +
                    (i*bw_per_point)**2.0)**0.5)
fo_enbw = arb_enbw(frst_ord, bw_per_point)
```

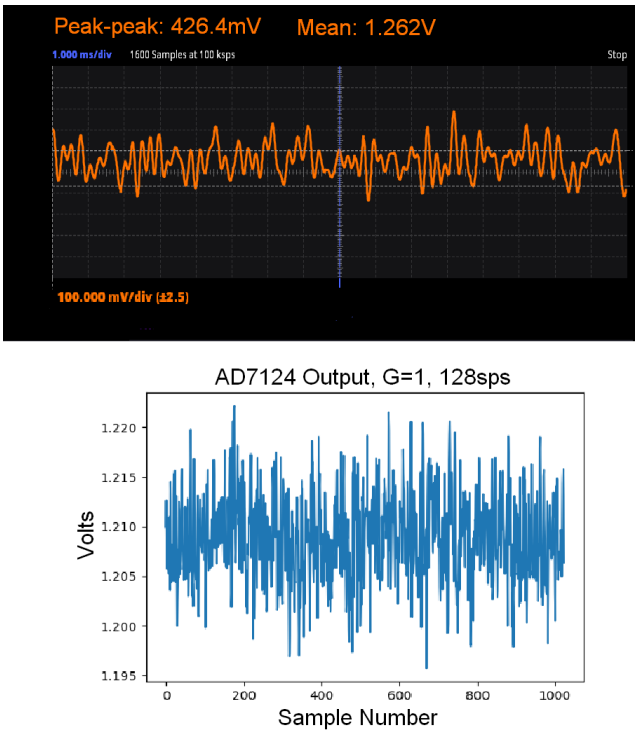

Fig. 19: *A 1mV/$\sqrt{Hz}$ noise band is driven into the AD7124-8 input. A qualitative reduction in noise is apparent; 426mV peak-to-peak noise at the ADC input results in approximately 25mV peak-to-peak noise at the ADC output. The 5.1mVRMS total output noise is close to the predicted 5.69mVRMS, given the 1mV/$\sqrt{Hz}$ noise density and 31Hz ENBW of the ADC's filter.*

This function can be used to calculate the ENBW of an arbitrary filter response, including the AD7124's internal filters. The frequency response of the AD7124 SINC4 filter, 128sps sample rate can be calculated similar to the previous 50/60Hz rejection filter example. The arb_anbw function returns a ENBW of about 31Hz.

The ADALM2000 noise generator can be used to validate this result. Setting the test noise generator to generate a band of 1000µV/$\sqrt{Hz}$ should result in a total noise of about 5.69mVRMS, and measured results are approximately 5.1mVRMS total noise. The oscilloscope capture of the ADC input signal is plotted next to the ADC output data, in Figure 19. Note the measured peak-to-peak noise of 426mV, while the ADC peak-to-peak noise is about 26mV. While such a high noise level is (hopefully) unrealistic in an actual precision signal chain, this exercise demonstrates that the ADC's internal filter can be relied on to act as the primary bandwidth limiting, and hence noise reducing, element in a signal chain.

Conclusion

Noise is a limiting factor in any signal chain; once noise contaminates a signal, information is lost. Before building a signal acquisition system, the application requirements must be understood, components selected accordingly, and the prototype circuit tested. This tutorial offers a collection of methods that accurately model and measure sensor and signal chain noise that can be used during the design and testing process.

The techniques detailed in this tutorial are, individually, nothing new. However, in order to achieve an adequate system, it becomes valuable to have a collection of fundamental, easy to implement, and low-cost techniques to enable signal chain modeling and verification. Even though industry continues to offer parts with increased performance, there will always be a certain limitation that one must be aware of. These techniques can not only be used to validate parts before building a mixed-mode signal chain, but also to identify design faults in an existing one.

Acknowledgements

- Jesper Steensgaard, who enabled/forced a paradigm shift in thinking about signal chain design, starting with the LTC2378-20.
- Travis Collins, Architect of Pyadi-iio (among many other things).
- Adrian Suciu, Software Team Manager and contributor to libm2k.

1. "Converter Connectivity Tutorial", https://wiki.analog.com/university/labs/software/iio_intro_toolbox, accessed 1 July, 2021.

2. Analog Devices Education Tools Repository https://doi.org/10.5281/zenodo.5105696

3. Pauli Virtanen, Ralf Gommers et al. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), 261-272.

4. "LTspice Simulator", https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html, accessed 1 July, 2021.

5. Smith, Steven W, The Scientist & Engineer's Guide to Digital Signal Processing, https://www.analog.com/en/education/education-library/scientist_engineers_guide.html, accessed 1 July, 2021.

6. Man, Ching, "Quantization Noise: An Expanded Derivation of the Equation, SNR = 6.02 N + 1.76", https://www.analog.com/media/en/training-seminars/tutorials/MT-229.pdf, accessed 1 July, 2021.

7. Kester, Walt, "Taking the Mystery out of the Infamous Formula, "SNR = 6.02N + 1.76dB"" Analog Devices Tutorial, 2009, https://www.analog.com/media/en/training-seminars/tutorials/MT-001.pdf, accessed 1 July, 2021.

8. "AD7124-8 Rev E" https://www.analog.com/media/en/technical-documentation/data-sheets/ad7124-8.pdf, accessed 1 July, 2021.

9. Charles R. Harris, K. Jarrod Millman, et al. Array programming with NumPy, Nature, 585, 357–362 (2020) DOI:10.1038/s41586-020-2649-2

10. "pyadi-iio: Device Specific Python Interfaces For IIO Drivers", https://wiki.analog.com/resources/tools-software/linux-software/pyadi-iio, accessed 1 July, 2021.

11. "Scopy", https://wiki.analog.com/university/tools/m2k/scopy, accessed 1 July, 2021.

12. "What is Libm2k?", https://wiki.analog.com/university/tools/m2k/libm2k/libm2k, accessed 1 July, 2021.

13. "LTC2057/LTC2057HV High Voltage, Low Noise Zero-Drift Operational Amplifier", https://www.analog.com/media/en/technical-documentation/data-sheets/2057f.pdf, accessed 1 July, 2021.

14. Kester, Walt, "Oversampling Interpolating DACs", Analog Devices Tutorial, 2009, https://www.analog.com/media/en/training-seminars/tutorials/MT-017.pdf, accessed 1 July, 2021.

15. Ruscak, Steve and Singer, L, "Using Histogram Techniques to Measure ADC Noise" Analog Dialogue, Volume 29, May, 1995. https://www.analog.com/en/analog-dialogue/articles/histogram-techniques-measure-adc-noise.html, accessed 1 July, 2021.

Modernizing computing by structural biologists with Jupyter and Colab

Blaine H. M. Mooers[‡§¶‖*]

Abstract—Protein crystallography produces most of the protein structures used in structure-based drug design. The process of protein structure determination is computationally intensive and error-prone because many software packages are involved. Here, we attempt to support the reproducibility of this computational work by using Jupyter notebooks to document the decisions made, the code, and selected output. We have made libraries of code templates to ease running the crystallography packages in Jupyter notebooks when editing them with JupyterLab or Colab. Our combined use of GitHub, snippet libraries, Jupyter notebooks, JupyterLab, and Colab will help modernize the computing done by structural biologists.

Index Terms—literate programming, reproducible research, scientific rigor, electronic notebooks, JupyterLab, Jupyter notebooks, computational structural biology, computational crystallography, biomolecular crystallography, protein crystallography, biomolecular structure, biomedical research, protein*drug interactions, RNA*drug interactions, molecular graphics, molecular visualization, scientific communication, molecular artwork, computational molecular biophysics

Introduction

Structural biologists study the molecular structures of proteins and nucleic acids to understand how they function in biology and medicine. The underlying premise of the field is that molecular function follows molecular form. More precise aliases for these scientists include molecular structural biologists, structural biochemists, and molecular biophysicists. Some of the methods used to determine the near-atomic resolution molecular structures include molecular modeling, X-ray crystallography, nuclear magnetic resonance (NMR), and cryo electron microscopy (cryo-EM). These scientists often use the molecular structures of these large biomolecules to design small-molecule drugs for improved therapies. As a result, structural biology plays a vital role in drug discovery and development, and many structural biologists work in the pharmaceutical industry. Those in academia in the United States generally have their work funded by the National Institutes of Health, the National Science Foundation, the Department of Defense, the Department of Energy, or one of several disease oriented medical foundations.

Structural biology is at the intersection of biochemistry, molecular biology, molecular biophysics, and computer science. Structural biologists have diverse backgrounds and varying levels of experience with computer programming ranging from minimal to very advanced. Several decades ago, the barriers to entry into the field included expertise with running command-line-driven programs and the ability to write programs to meet data analysis needs not met by existing software packages. However, these barriers have been lowered over the past two decades by the widespread availability of GUI-driven software that is often free for academics (e.g., CCP4 [Winn11], Phenix [Lieb19], CNS [Brun98], ATSAS [Mana21], BioXTAS [Hopk17], CCPEM [Burn17]). As a result, biologists, who often have little formal training in computing, have become the largest component of the field.

Computing is involved in the six or more steps from structural data acquisition to publication. Several alternate software packages are often available for each step. Different combinations of these alternatives lead to a combinatorial explosion of possible workflows. In some situations, workers have set up software pipelines for some of the steps. However, these pipelines are difficult to transfer or have trouble with the challenging samples that cannot yet be handled without human intervention. The current heterogenous computing environment makes the computational work vulnerable to errors in the tracking of input and output files. Storing the code and outputs for some of the steps in Jupyter notebooks would be one way to reduce this vulnerability [Kluy16], [Gran21]

To ease crystal structure determination in Jupyter, we made libraries of code templates for crucial programs. We formatted the libraries for two extensions of JupyterLab that provide access to code snippets. One extension (jupyterlab-snippets) displays the snippets in nested pull-down menus [jLsnip]. The other extension (elyra-code-snippet-extension) uses a search box to locate the desired snippet [ELSN] (also see the blog post by Luciano Resende [Rese20]). The user can easily add new code snippets to both systems.

We also ported the libraries to Google Colaboratory or Google Colab or just Colab [Carn18], [Cola21]. Colab is an integrated development environment (IDE) for running Jupyter notebooks on the Google Cloud Platform (GPC). Colab was designed to promote the adaptation of deep learning software to new problems and facilitate collaborative computing. Colab is a free service that provides a temporary instance of a Linux operating system with access to one K80 GPU through a Jupyter notebook. Access to

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

TPUs is also available. The access is terminate after long periods of inactivity or a 12-hour time limit, whichever is reached first. The time limit can be extended with a small subscription fee.

Colab comes with some deep learning software and the Python scientific computing stack including SciPy [SciP20]. Colab spares the user of the maintenance of the hardware and the operating system software. Colab can also serve as a test platform for software on Linux when one's primary computing environment is a Mac or Windows. Colab also eases collaborative work and provides a uniform computing environment for classes and workshops. The use of Colab requires that the user have a Google Drive account for storing software, Jupyter notebooks, and data files.

The user can install additional Python and other packages including structural biology software, provided the user has the required software licenses. This software installation step requires extra time at the start of the Colab session because most structure biology software packages have numerous dependences. To accelerate this setup step, we provide notes and code snippets for the installation of this software in the Jupyter notebook that serves as the carrier of the snippet libraries on Colab. The user can install the required software in several minutes. Although the software is deleted automatically from Google Cloud at the end of session, the software can be stored on the user's Google Drive for faster reinstallation.

Methods

We created snippet libraries for each structural biology package to support structural biology computations in Jupyter and Colab. Any particular workflow is unlikely to require all of the libraries. For example, a beginner's workflow is unlikely to use CCTBX, a library of Python wrapped C++ routines for building molecular structure determination software. Likewise, a cryo-EM workflow will not need XDS, a package for processing X-ray diffraction images. We created a GitHub site for each library to ease the downloading of only those libraries that interest users (Table **??**). This modularization of the project should ease the correction and augmentation of individual libraries as the extensions, and structural biology software packages evolve. We only provided libraries for JupyterLab because the Jupyter Project plans to phase out support for the Jupyter Notebook software. Among the several alternative extensions for code snippets in JupyterLab, we choose jupyterlab-snippets [jLsnip] and Elyra [Elyra] because these two extensions are actively maintained and have different features. We also support a snippet library for Jupyter notebooks on Google Colab as described below because Colab provides access to GPUs, which can accelerate some of the larger computational tasks.

The jupyterlab-snippets extension

The jupyterlab-snippets extension adds a snippet menu to the JupyterLab menu bar. The user accesses the snippets through a cascading pulldown menu. Each snippet resides in a separate plain text file without any formatting. This feature dramatically eases adding new snippets by users and eases keeping the snippets under version control. The snippets are stored in the Jupyter data directory (which is found by entering `jupyter --path`; it is in ~/Library/Jupyter/snippets on Mac OS). Each snippet library is stored in a separate subfolder, which appears on the menu bar as a part of a cascading pulldown menu (Figure 1).

We clustered snippets into categories. Each category has a cascading submenu. Clicking on a snippet name in the submenu triggers its insertion into the current cell in the notebook. The nested menu hierarchy serves well the user who is familiar with the content of the snippet libraries.

Fig. 1: *Cascading pull-down menu for the Jupyter categories of the jupyterlabpymolpysnips library.*

Like most other snippet extensions for Jupyter Notebook and JupyterLab, the jupyterlab-snippets extension does not support tab stops or tab triggers. These are common features of snippet libraries for most text editors and IDEs that accelerate the editing of parameter values in snippets. The tab stops are particularly valuable because they direct the user to sites that may need changes in their parameter values, and they guide the user to all of the site to ensure that none are overlooked. The overlooking of parameter values that require changing can be a major source of bugs. The tab triggers are also often mirrored, so a change at one instance of the same parameter will be propagate automatically to other identical instances of the parameter. To compensate for the lack of tab triggers, we include a second copy of the code in the same snippet but in a comment and with the tab triggers marked with curly braces and numbers (Figure 2). The user uses the code in the comment to direct their editing of the active code. The user can delete the commented out comment when they have finished editing. Separate versions of the libraries were made with commented out code. These versions are distinguished by having "plus" appended to their names.

The elyra-code-snippet extension

A menu icon labeled with </> provides access to snippets in the elyra-code-snippet-extension system. After the icon is clicked, the snippets appear in the left margin of the JupyterLab GUI. Snippets from all libraries appear in alphabetical order. The user can scroll through the list of snippets. Hovering the mouse cursor over the snippet's name triggers the display of a description of the snippet.

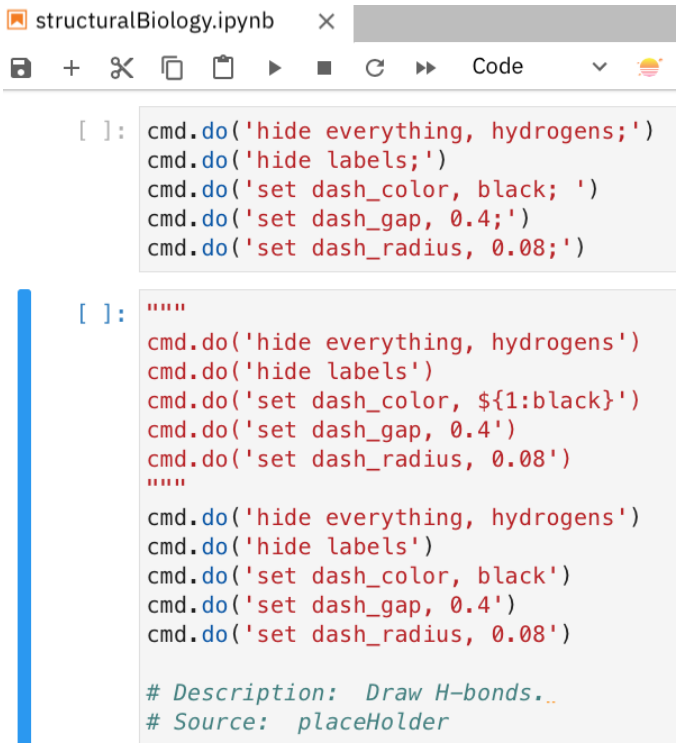

Fig. 2: *Comparison of active code in the bottom code block and the commented out code above the active code from a code snippet. The commented lines of code serve as guides for editing because they have curly braces marking sites to be considered for editing. The commented lines of code compensate for the absence of tab stops.*

Alternatively, the user can enter a search term in the search box at the top of the menu to reduce the list of snippets. The search terms can be part of a snippet name or a tag stored with each snippet.

A tag icon displays all of the available tags in the snippets as separate icons. The user can select tags to be used to choose snippets by clicking on the icons.

Each snippet is displayed with several icons (Figure 3). A triangular toggle can trigger the display of the snippet in a textbox. A pencil icon enables the editing of the code. Other icons enable copying the code to the clipboard, inserting code into the current cell in the notebook, and deleting the snippet.

A plus sign in the upper-righthand corner opens a GUI for the creation of a new snippet. The GUI occupies a new tab in the window that houses the Jupyter notebooks. The GUI has a text box for each kind of metadata: name, description, tags, language, and the snippet code. There is a save button at the bottom to add the new snippet to the current library.

Each snippet is stored in a separate JSON file. Each JSON file has the snippet code plus several rows of metadata, including a list of tags and the programming language of the snippet. The latter provides a sanity check. For example, an attempt to insert a C++ snippet into a notebook with an active Python kernel will trigger the opening of a window with a warning.

All of the snippets reside in the folder url{JUPYTER_DATA/metadata/code-snippets}. This is the directory url{~/Library/Jupyter/metadata/code-snippets} on the Mac. There are no subfolders for individual snippet libraries, unlike the jupyterlab-snippets extension. The snippets from multiple libraries are stored together in the code-snippets folder. The tag system can be used to select all snippets from one library. The tag system serves well the user who is not familiar with the content of the installed libraries. The user can download the snippets from GitHub as zip file and then uncompress this file and move the snippet files to the final destination.

Colab snippet library

The Colab snippet system resembles the Elyra snippet system in that the snippets appear in a menu to the left of the notebook and that search terms in a search box retrieve snippets. However, the Colab system differs from the Elyra system ins that the snippets are stored in one or more Jupyter notebooks. The user's Google Drive stores the notebook of snippets. The user enters the url for the notebook in a the Tools --> Settings --> Site --> Custom Snippet Notebook URL. Multiple URLs for multiple notebooks can be entered at one time. The user logs out of Colab and upon logging in again to install the snippets. The user will see the newly added snippets in the left margin after opening the snippet menu by clicking on the </> icon.

Each snippet had a markdown cell followed by a code cell. The markdown cell contained the name of the snippet, a description of what the snippet does, and the structural biology software. These features are searched in the search box to narrow the list of snippets to inspect for selection.

The first snippet in each notebook provided the steps for installing the software on Colab. The markdown cell listed these installation steps. Then a series of code snippets contained the code for carrying out the steps. This installation snippet was the only one in a notebook that contained more then one code snippet.

A search box at the top of the list of snippets is used to recover a snippet (Figure 5. The user enters a snippet name in the search box to display the snippet and its documentation. The user hits the 'Install' button to install the snippet's code at the current position in the notebook. Unlike the Elyra snippets which insert a whole snippet into one code block, a Colab snippet can have multiple code blocks that are inserted into the notebook at the current position of the mouse cursor. One snippet can have different types of code blocks. For example, the snippet in Figure 5 has a three blocks of Python code, two blocks of shell commands, and two blocks of bash cell magics with multiple lines of bash commands.

The list snippet for a library will print in a table below the current cell a list of the snippets in the library and a brief description. This table is stored in a pandas DataFrame that can be searched with the pandas search function. This table can also be searched for key terms with the search function in the notebook. The code block and output can be hidden by clicking on the three blue dots on the left margin of the cell.

Notebooks on Colab open very quickly, but the user must reinstall their software on each login. We ease this annoying task by supplying the complete chain of installation steps. For example, the installation of the molecular graphics program PyMOL requires seven code blocks of different types. Some involve the use of curl, and others use the conda package management system. We include all steps in one snippet, which is uniquely possible with the snippet system for Colab (Figure 5). The user only has to select one snippet and then run each code block in succession.

The use of Colab requires that the user has a Google account and a Google Drive. Many structural biologists already have both.

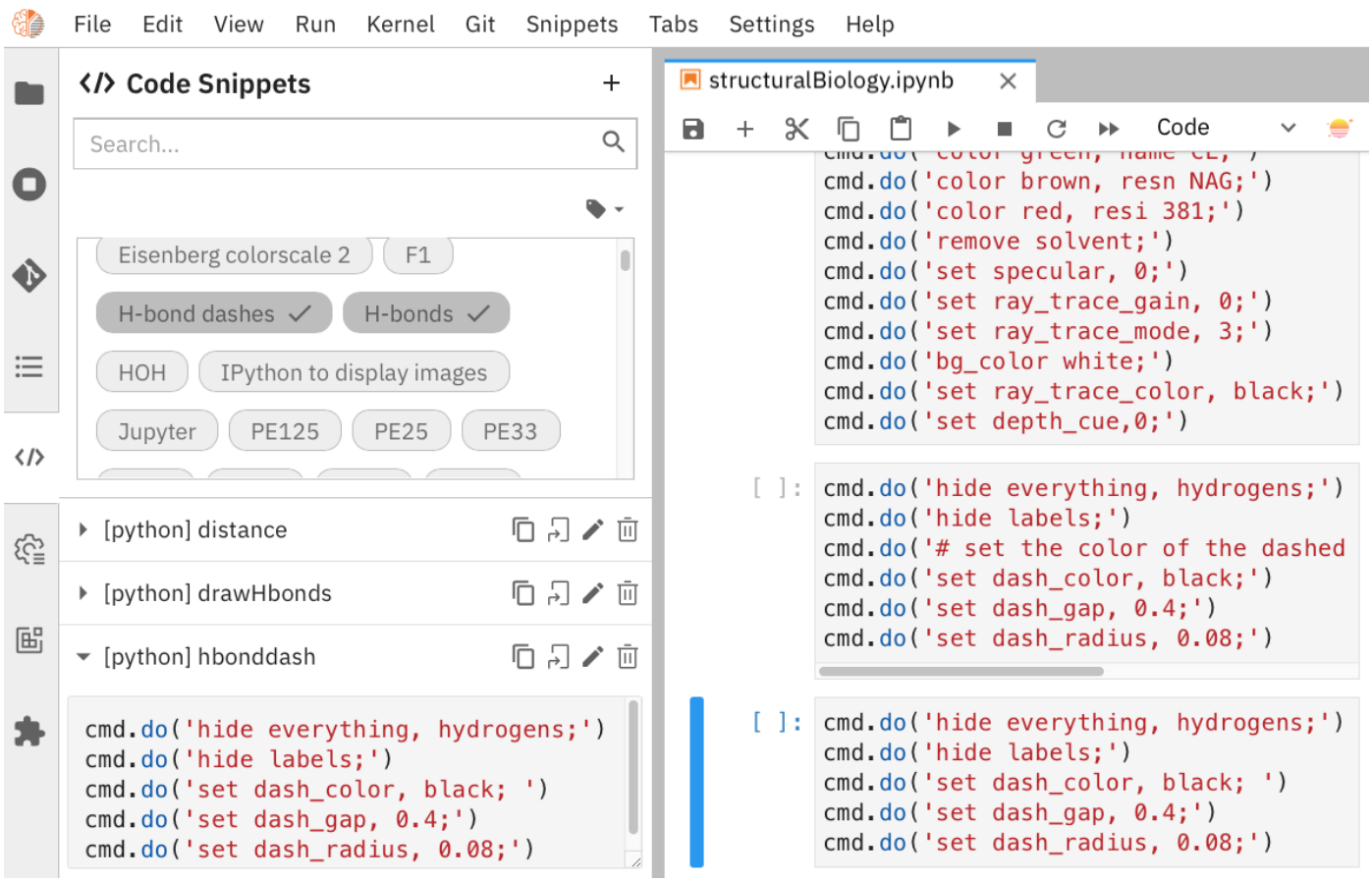

Fig. 3: *The GUI from the elyra-code-snippet extension for accessing code snippets is shown on the left. A preview of the* hbonddash *snippet is shown in the lower left. A Jupyter notebook with the inserted the* hbonddash *snippet is shown on the right.*

Notebooks with sample workflows

We created a library of Jupyter Notebooks with sample workflows. This library of notebooks is only representative and not exhaustive because the combinatorial explosion of possible workflows makes covering all workflows impractical. These notebooks can serve as templates for the creation of new notebooks and are available on our GitHub site [MLGH].

Availability of the snippet libraries

We have shared these libraries on GitHub [MLGH]. Each library is also archived in zenodo.

Results

We describe here a set of libraries of code templates to support computational crystallography in Jupyter Notebooks on the cloud and on local computers. The libraries and notebooks can be loaded on and run on Google Colab where the user can share the notebook with collaborators or gain access to GPUs and TPUs. The user uploads the libraries and notebook to their Google Drive account and accesses the notebook from Colab. The storage of the libraries and notebooks on Google Drive persists between logins to Google Colab, but the crystallographic software must be reinstalled on each use of Colab. These libraries are installed only once; however, the crystallographic software must be reinstalled upon each login. We describe below installation scripts in the form of snippets that can be quickly run at the top of a Notebook to minimize the effort required to re-install the software. Another limitation of the Colab snippet system is that snippets from all libraries are stored in one pool and have to be accessed by either scrolling through a long list or by entering the snippet name in a search box. We addressed this limitation with a snippet for each library that prints a list of the available snippets with a short description. This list can span more than the length of a paper, but it can be collapsed to hide it or can be deleted when no longer needed. After the snippet is pulled out of the list by the search box, more detailed documentation about the snippet is displayed. Next, we describe the content of each library with representative output in the Colab notebook.

Structure determination and refinement workflows with Phenix

A team of professional software developers based at the Berkeley-Lawrence National Laboratory (BLNL) develops the Phenix software to refine protein crystal structures determined from X-ray diffraction data [Adam02]. The project includes several collaborators located around the world who develop auxiliary components of the package. Phenix uses Python to interface with the Computational Crystallography Tool Box (CCTBX), which is written in C++ for speed [Gros02]. CCTBX is also wrapped in Python and can be imported at as module. While Python eases the use of CCTBX, mastery of CCTBX requires at least an intermediate level of Python programming skills. On the other hand, Phenix is easy to use via the command line or a GUI and has become of the most popular software packages for biological crystallography.

The Phenix project greatly eased the incorporation of simulated annealing into crystal structure refinement by hiding the

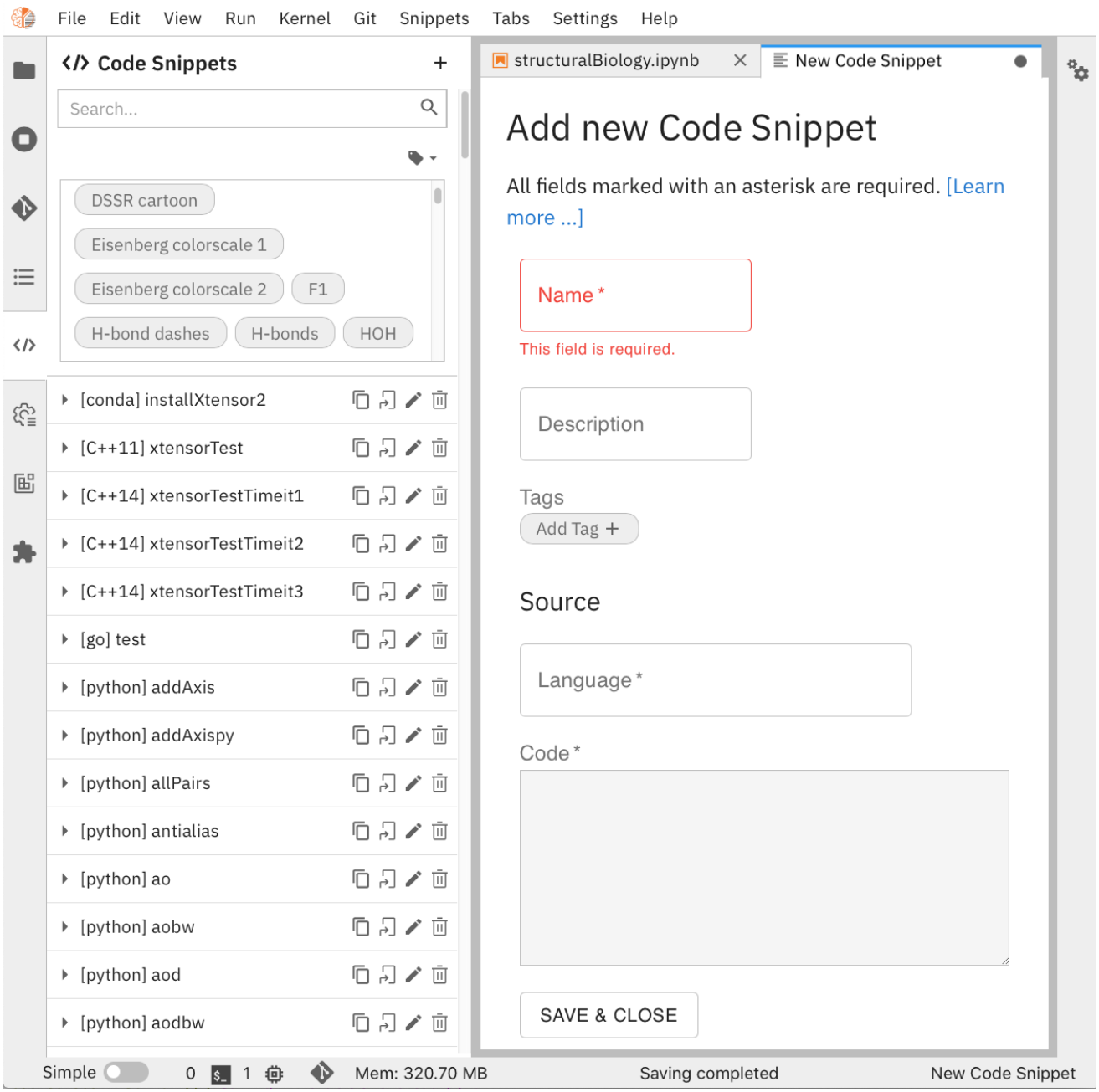

Fig. 4: *The GUI from elyra-code-snippet extension for the creation of new snippets. The Learn more link takes the user to the documentation on Read-the-docs.*

tedious preparation of the required parameter files from the user. Simulated annealing involves molecular dynamics (MD) simulation at high temperatures to move parts of a molecular model out of local energy minima and into conformations that fit the experimental data better. Twenty minutes of applying simulated annealing to an early model that still has numerous errors can significantly improve the model while saving the user a day or more of the tedious manual rebuilding of the molecular model. The PDB file does not have sufficient information about chemical bonding for MD simulations. The molecular dynamics software that carries out the simulated annealing requires two parameter files and the coordinate file. The preparation and debugging of the parameter files manually takes many hours, but Phenix automates this task.

More recently, Phenix has been extended to refine crystal structures with neutron diffraction data and for structure determination and refinement with cryo-EM data [Lieb19]. The addition of support for cryo-EM help address the recent need for the ability to fit atomic models to cryo-EM maps that have recently become available at near atomic resolution because of the dramatic improvements in detector technology. Users can interact with Phenix via a GUI interface or the command line, as mentioned before, but users can also use PHIL, domain-specific language scripting language for more precise parameter settings for Phenix. In addition, users can use the `phenix.python` interpreter. Unfortunately, the phenix.python interpreter is still limited to Python2, whereas CCTBX has been available for Python3 for over a year.

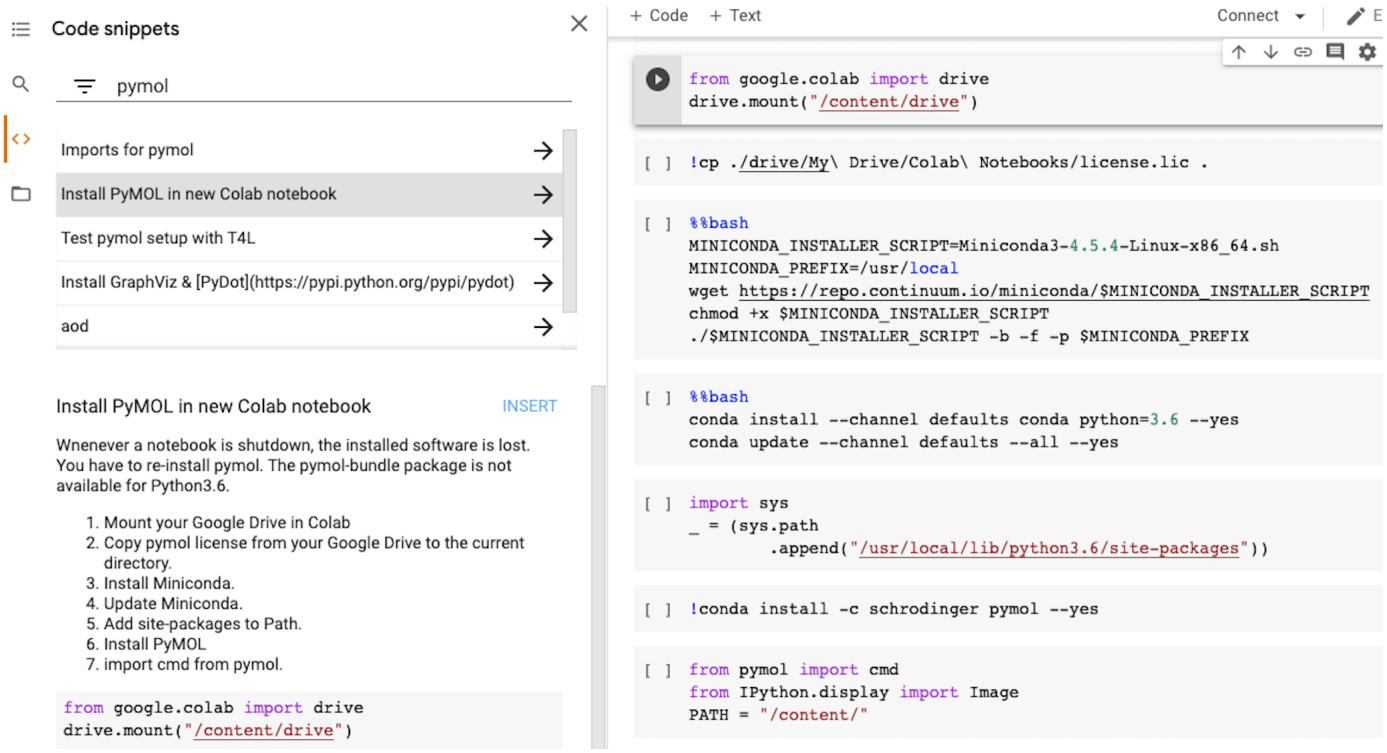

Fig. 5: *Code snippet for installing PyMOL on Colab. The <> icon opens a menu on the left side that lists all of the snippets. The search term 'pymol' was used to reduce the list of candidate snippets. The highlighted snippets name 'Install PyMOL is new Colab notebook'. Selecting this snippets opens the snippet below. The snippet description is displayed followed by the seven blocks of code. The description includes the seven steps for installing the molecular graphics programs. Clicking with the mouse cursor on 'INSERT' in blue inserts the code into in the cells on the notebook on the fight.*

Jupyter Lab and its extensions are also best run with Python3. The most practical approach to using Phenix in Jupyter Lab is to invoke Phenix by utilizing the shell rather than using Python. For example, the command shown below invokes statistical analysis of the B-factors in a Protein Data Bank (PDB) file by using one line of code in the shell. The PDB file uses a legacy, fixed-format file for storing the atomic coordinates and B-factors of crystal structures. The B-factors are a measure of the atomic motion, statistical disorder, or both in individual atoms in a protein structure. The PDB file format was defined and popularized by the Protein Data Bank, a repository for atomic coordinates and structural data that has over 170,000 entries from around the world. The PDB was started in 1972 and unified with the branches in Japan and Europe in 2003 as the wwPDB [Berm03]. The wwPDB continues to play a central role in promoting the principles of open science and reproducible research in structural biology.

Since 2019, the wwPDB requires the PDBx/mmCIF format for new depositions [Adam19]. Many structural biology software packages now have the ability to read files in the PDBx/mmCIF format.

```
!phenix.b_factor_statistics 1lw9.pdb
```

The output form this command is printed below the cell that invokes the command. Some of the output is shown below.

```
Starting phenix.b_factor_statistics
on Wed Jun  2 04:49:01 2021 by blaine

Processing files:

  Found model, /Users/blaine/pdbFiles/1lw9.pdb

Processing PHIL parameters:

  No PHIL parameters found

Final processed PHIL parameters:

  data_manager {
    model {
      file = "/Users/blaine/pdbFiles/1lw9.pdb"
    }
    default_model = "/Users/blaine/pdbFiles/1lw9.pdb"
  }


Starting job
Validating inputs
                 min     max    mean <Bi,j>   iso aniso
   Overall:    6.04 100.00  24.07    N/A  1542     0
   Protein:    6.04 100.00  23.12    N/A  1328     0
   Water:      9.98  55.93  30.47    N/A   203     0
   Other:     14.11  35.47  21.10    N/A    11     0
   Chain  A:   6.04 100.00  24.07    N/A  1542     0
   Histogram:
       Values      Number of atoms
     6.04 - 15.44        309
    15.44 - 24.83        858
    24.83 - 34.23        187
    34.23 - 43.62         78
    43.62 - 53.02         32
    53.02 - 62.42         16
    62.42 - 71.81          8
    71.81 - 81.21          6
    81.21 - 90.60          2
    90.60 - 100.00        46

Job complete
usr+sys time: 1.92 seconds
wall clock time: 2.93 seconds
```

There are several dozen commands that can be run via the shell and return useful output that can be captured in one Jupyter Notebook rather than in dozens of log files. The output can be copied and pasted into a new cell and then reformatted in markdown as a table or the copied output be used as input data to make a plot with matplotlib. While these are basic data science tasks, they are intimidating to new users of Jupyter and some of the details are easy for more experienced users to forget. To overcome this problem, we supply snippets that demonstrate how to transform the output and that can be used as templates by the users.

These commands are becoming harder to find as the on-line documentation has been migrating to serving only the GUI interface. The bash script files that run the Phenix commands can be found on Mac OSX by running the following command:

```
!ls /Applications/phenix-*/build/bin/phenix.\*
```

These shell scripts invoke Python scripts that capture the command line arguments and pass them to the Phenix Python interpreter. This Python script files can be found on Mac OSX by running the following command:

```
!ls /Applications/phenix-1.19.2-4158/modules/phenix/phe
```

Molecular graphics with PyMOL

The end result of the crystal structure refinement in Phenix is a set of atomic coordinates. They can be displayed in one of the many available molecular graphics programs like PyMOL [PyMO21]. If PyMOL is available in the current Python environment, PyMOL's Python API can be accessed by importing the *cmd* class. In addition, it is useful to import the *Image* class from IPython to be able to upload images written to disk by PyMOL.

```
from pymol import cmd
from IPython.display import Image
```

After installing PyMOL in Colab as outlines in Figure 5 and the PyMOL snippet library, the *T4L* snippet was inserted into a Colab notebook and executed. The snippet includes the IPython command that was used to upload the image into the Notebook as shown in Figure 6.

There are several other methods of importing images including using Markdown or HTML code.

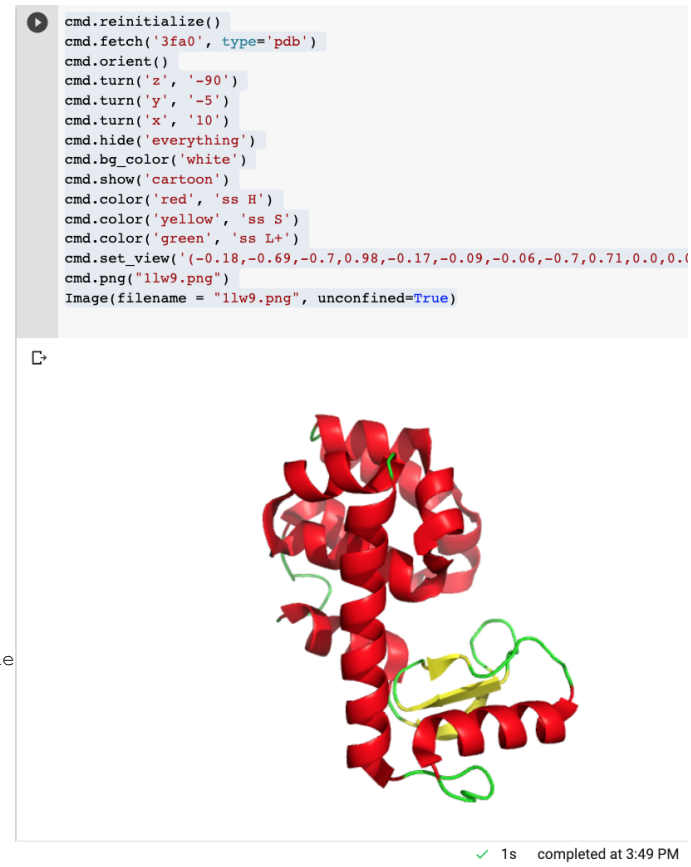

Fig. 6: *The code of the* T4L *snippet inserted into a code block in Colab. .*

Discussion

Amazon introduced the first cloud computing service in 2006; there are now over 200 services. These services have the advantage of providing access to computer hardware and software. These services can lower barriers for those labs that have limited access to computer hardware or that have trouble with installing software. Many of these services supply disk spaces and access to CPUs, GPUs, and sometimes TPUs. Access to basic services is often free with additional services, computing power, and disk space being available for a modest fee. In principle, consumer computers could be used as an interface for doing all manner of crystallographic computing on the cloud.

Why Colab?

Colab was developed internally and first released for public use in 2018. Numerous research papers in the physical and medical sciences have been published that used Colab. Google Colab provides fast and easy access for users with a Google account and Google drive, so many workers in crystallography already have the prerequisites. Many readers are also familiar with Jupyter Notebooks (10 million shared on GitHub as of early 2021). Jupyter Notebooks can be loaded onto Google Drive and then opened in Colab. Colab is a specialized IDE for editing Jupyter Notebooks. the Colab interfaces has more features that the easy-to-use nteract IDE, but fewer features than JupyterLab. Colab provides almost instant loading of specific Jupyter notebooks but at the cost of needing reinstall the software used in a notebook upon logging in again or after a 12-hour session. The first point lower the barrier to resuming work while the second point can be addressed by including the code for installing the required software at the head of the notebook.

Microsoft has stopped supporting its Azure Notebook and has asked users to migrate to several alternative approaches. One approach is to use of Visual Studio Code (VSC) rather to JupyterLab to edit and run Jupyter notebooks locally and on Microsoft's cloud service. VSC is an advanced text editor that has stronger support for code snippets because it supports the use of tab triggers and tab stops, two important features that are missing from Colab, JupyterLab, and the Classic Jupyter Notebook. However, VSC is so feature-rich that it can be over-whelming for some beginning users. To support a wider group of users, we developed the libraries for Google Colab. We plan to develop libraries for editing Jupyter Notebooks in VSC.

What is new

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

logins. These templates include the code for installing the software required for crystallographic computing. These installation templates save time because the installation process involves as many as seven operations that would be difficult to remember. Once the user adds the installation code to the top of a given notebook, the user only needs to rerun these blocks of code upon logging into Colab to be able to reinstall the software. The user can modify the installation templates to install the software on their local machines. Examples of such adaptations are provided on a dedicated GitHub webpage. The template libraries presented here lower an important barrier to the use of Colab by those interested in crystallographic computing on the cloud.

Relation to other work with snippet libraries

To the best of our knowledge, we are the first to provide snippet libraries for crystallographic computing. This library is among the first that is domain specific. Most snippet libraries are for programming languages or for hypertext languages like HTML, markdown and LaTeX. The average snippet also tends to be quite short and the size of the libraries tends to be quite small. The audience for these libraries are millions of professional programmers and web page developers. We reasoned that this great tool should be brought to the aid of the thousands of workers in crystallography.

The other area where domain specific snippets have been provided is in molecular graphics. The pioneering work on a scripting wizard provided templates for use in the molecular graphics program RasMol [Hort99]. The conscript program provided a converter from RasMol to PyMOL [Mott10]. Language converters for translating code between the leading molecular graphics programs would allow users to more easily find and use the optimal molecular graphics program for the task at hand.

We also provided snippets for PyMOL, which has 100,000 users, for use in text editors [Moo21a] and Jupyter notebooks [Moo21b]. The former support tab triggers and tab stops; the latter does not.

The libraries have to be molecular graphics program specific because molecular graphics programs have been written in a range of programming languages. The user issues the commands in either in a general programming language like Python or a a domain specific language (DSL) like pml. It would cause confusion to mix snippets from multiple languages. To counter this growing tower of babel, the OpenStructure initiative was formed [Bias13].

We have also worked out how to deploy this snippet libraries in OnDemand notebooks at High-Performance Computing centers. These notebooks resemble Colab notebooks in that JupyterLab extensions cannot be installed. However, they do not have any alternate support for accessing snippets from menus in the GUI. Instead, we had to create IPython magics for each snippet that load the snippet's code into the code cell. This system would also work on Colab and may be preferred by expert users because the snippet names used to invoke magic are under autocompletetion. That is, the user enters the start of a name and IPython suggests the remainder of the name in a pop-up menu. We offer a variant library that inserts a commented out copy of the code that has been annotated with the sites that are to be edited by the user.

Opportunities for Interoperability

The set of template libraries can encourage synergistic interoperability between software packages supported by the snippet libraries. That is the development of notebooks that use two or more software packages and even programming languages. More general and well-known examples of interoperability include the Cython packages in Python that enable the running of C++ code inside Python, the reticulate package that enables the running of Python code in R , and the PyCall package in Julia that enables the running of the Python packages in Julia. The latter package is widely used to run matplotlib in Julia. Interoperability already occurs between the CCP4, clipper, and CCTBX projects and to a limited extent between CCTBX and PyMOL, but interoperability could be more widespread if the walls around the software silos were lowered. The snippet libraries provided here can prompt interoperability on Colab by their proximity on Colab.

Polyglot snippets

The unique feature of the Colab snippets is that a given snippet can contain multiple cells. The cells can be a mix of markdown (text cells) and code cells. The cells can also use a mix of programming languages invoked by different cell magics. Cell magics are an alternate method to kernels for switching between programming languages. The code for defining various cell magics are included in our snippet library. The supported compiled programming languages include C, C++, Julia, and Fortran2008. The bash cell magic is built into Colab. This ability to two or more programming languages in one snippet leads to polyglot snippets. Some operations involving two or more programming languages need to be executed sequentially. These can be best grouped together in one snippet. This feature of polyglot snippets save time because the user does not have to reinvent the workflow by finding and inserting into the notebook a series of snippets.

Ubiquitous computing platform on the cloud

Colab provides the user with a ubiquitous instance of Ubuntu. Colab is accessed by opening Jupyter Notebooks stored on the users' Google Drive account. Colab can be accessed from devices that can access the Google Drive account. The opening of the Colab instance is rapid in contrast to the Binder service where the building of a new Ubuntu instance requires a wait of many minutes. In addition, the Colab session remains active for up to 12 hours on the free plan and longer on paid plans whereas a Binder instance closes after ten minutes of inactivity. Binder is an open-source project while Colab is a closed source project. Colab maintains the Ubuntu operating system so the user does not need to spend time on software updates.

Acknowledgements

This work is support in part by these National Institutes of Health grants: R01 CA242845, P20 GM103640, P30 CA225520.

REFERENCES

[Adam02] P. D. Adams, R. W. Grosse-Kunstleve, L.-W. Hung, T. R. Ioerger, A. J. McCoy, N. W. Moriarty, R. J. Read, J. C. Sacchettini, N. K. Sauter, and T. C. Terwilliger. *PHENIX: building new software for automated crystallographic structure determination*, Acta Cryst. D58(11):1948–1954, November 2002. doi: 10.1107/S0907444902016657

[Adam19] P. D. Adams, P. V. Afonine, K. Baskaran, H. M. Berman, J. Berrisford, G. Bricogne, D. G. Brown, S. K. Burley, M. Chen, Z. Feng, C. Flensburg, A. Gutmanas, J. C. Hoch, Y. Ikegawa, Y. Kengaku, E. Krissinel, G. Kurisu, Y. Liang, D. Liebschner, L. Mak, J.L Markley, N. W. Moriarty, G. N. Murshudov, M. Noble, E. Peisach, I. Persikova, B. K. Poon, O. V. Sobolev, E. L. Ulrich, S. Velankar, C. Vonrhein, J. Westbrook, M. Wojdyr, M. Yokochi, and J. Y. Young. *Announcing mandatory submission of PDBx/mmCIF format files for crystallographic depositions to the Protein Data Bank (PDB)*, Acta Crystallographica Section D: Structural Biology, 75(4):451-454, April 2019. doi: 10.1107/S2059798319004522

[Beg21] M. Beg, J. Belin, T. Kluyver, A. Konovalov, M. Ragan-Kelley, N. Thiery, and H. Fangohr. *Using Jupyter for reproducible scientific workflows*, Computing Sci. & Eng., 23(2):36-46, April 2021. doi: 10.1109/MCSE.2021.3052101

[Berm03] H. Berman, K. Hendrick, and H. Nakamura. *Announcing the worldwide Protein Data Bank*, Nature Structural & Molecular Biology, 10(12):980, December 2003.

[Bias13] M. Biasini, T. Schmidt, S. Bienert, V. Mariani, G. Studer, J. Haas, N. Johner, A. D. Schenk, A. Philippsen, and T. Schwede. *OpenStructure: an integrated software framework for computational structural biology*, Acta Cryst. D69(5):701–709, May 2013. doi: 10.1107/S0907444913007051

[Brun98] A.T. Br{"u}nger, P.D. Adams, G.M. Clore, W.L. Delano, P. Gros, R.W. Grosse-Kunstleve, J.-S. Jiang, J. Kuszewski, M. Nilges, N. S. Pannu, R. J. Read, L. M. Rice, T. Simonson, and G. L. Warren. *Crystallography & NMR system: A new software suite for macromolecular structure determination*, Acta Cryst. D54(5):905-921, May 1998. doi: 10.1107/S0907444998003254

[Burn17] T. Burnley, C.M. Palmer, and M. Winn. *Recent developments in the CCP-EM software suite*, Acta Cryst. D73(6):469-477, June 2017. doi: 10.1107/S2059798317007859

[Carn18] T. Carneiro, R. V. M. Da N{'o}brega, T. Nepomuceno, G.-B. Bian, V. H. C. De Albuquerque and P. P. Reboucas Filho. *Performance analysis of google colaboratory as a tool for accelerating deep learning applications*, IEEE Access 6:61677-61685, November 2018. doi: 10.1109/ACCESS.2018.2874767

[Cola21] https://colab.research.google.com

[ELSN] https://elyra.readthedocs.io/en/latest/user_guide/code-snippets.html

[Elyra] https://github.com/elyra-ai/elyra/blob/master/docs/source/getting_started/overview.md

[Godd18] T. D. Goddard, C.C. Huang, E.C. Meng, E.F. Pettersen, G.S. Couch, J. H. Morris, and T. E. Ferrin. *UCSF ChimeraX: Meeting modern challenges in visualization and analysis*, Protein Sci., 27(1):14-25, January 2018. doi: 10.1002/pro.3235.

[Gran21] B. E. Granger and F. Pérez. *Jupyter: Thinking and Storytelling With Code and Data*, Computing in Science & Engineering, 23(2):7-14, March-April 2021. doi: 10.1109/MCSE.2021.3059263

[Gros02] R. W. Grosse-Kunstleve, N. K. Sauter, N. W. Moriatry, P. D. Adams. *The Computational Crystallography Toolbox: crystallographic algorithms in a reusable software framework*, J Appl Cryst, 35(1):126-136, February 2002. doi: 10.1107/S0021889801017824.

[Hopk17] J.B. Hopkins, R. E. Gillilan, and S. Skou. *BioXTAS RAW: improvements to a free open-source program for small-angle X-ray scattering data reduction and analysis*, J. Appl. Cryst., 50(5):1545–1553, October 2017. doi: 10.1107/S1600576717011438

[Hort99] R. M. Horton. *Scripting Wizards for Chime and RasMol*, Biotechniques, 26(5):874-876, May 1999. doi: 10.2144/99265ir01

[Kluy16] T. Kluyver, B. Ragan-Kelley, F. P{'e}rez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team. *Jupyter Notebooks -- a publishing format for reproducible computational workflows*, In F. Loizides and B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas (pp, 87-90). doi: 10.3233/978-1-61499-649-1-87

[jLsnip] https://github.com/QuantStack/jupyterlab-snippets

[Lieb19] D. Liebschner, P. V. Afonine, M. L. Baker, G. Bunkóczi, V. B. Chen, T. I. Croll, B. Hintze, L.-W. Hung, S. Jain, A. J. McCoy, N.W. Moriarty, R. D. Oeffner, B. K. Poon, M. G. Prisant, R. J. Read, J. S. Richardson, D. C. Richardson, M. D. Sammito, O. V. Sobolev, D. H. Stockwell, T. C. Terwilliger, A. G. Urzhumtsev, L. L. Videau, C. J. Williams, and P. D. Adams. *Macromolecular structure determination using X-rays, neutrons and electrons: recent developments in Phenix*, Acta Cryst., D75(10):861-877, October 2019. doi: 10.1107/S2059798319011471

[Mana21] K. Manalastas-Cantos, P. V. Konarev, N. R. Hajizadeh, A. G. Kikhney, M. V. Petoukhov, D. S. Molodenskiy, A. Panjkovich, H. D. T. Mertens, A. Gruzinov, C. Borges, M. Jeffries, D. I. Svergun, and D. Franke. *ATSAS 3.0: expanded functionality and new tools for small-angle scattering data analysis*, J. Appl. Cryst., 54(1):343–355, February 2021. doi: 10.1107/S1600576720013412

[Mott10] S. E. Mottarella, M. Rosa, A. Bangura, H. J. Bernstein, and P. A. Craig. *Conscript: RasMol to PyMOL script converter*, Biochem. Mol. Biol. Educ., 38(6):419-422, November 2010. doi: 10.1002/bmb.20450

[MLGH] https://github.com/MooersLab

[Moo21a] B. H. M. Mooers and M .E. Brown. *Templates for writing PyMOL scripts*, Pro. Sci., 30(1):262-269, January 2021. doi: 10.1002/pro.3997

[Moo21b] B. H. M. Mooers. *A PyMOL snippet library for Jupyter to boost researcher productivity*, Computing Sci. & Eng., 23(2):47-53, April 2021. doi: 10.1109/mcse.2021.3059536

[Nguy17] H. Nguyen, D. A. Case, and A.S. Rose. *NGLview--interactive molecular graphics for Jupyter notebooks*, Bioinformatics, 34(7):1241-1242, April 2017. doi: 10.1093/bioinformatics/btx789

[PyMO21] https://pymol.org/2/

[Rese20] https://blog.jupyter.org/reusable-code-snippets-in-jupyterlab-8d75a0f9d207

[SciP20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. {van der Walt}, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J.Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. {VanderPlas}, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. {van Mulbregt}, Paul and {SciPy 1.0 Contributors}. *{{{SciPy} 1.0: Fundamental Algorithms for Scientific Computing in Python}}*, Nature Methods, 17(3):261-272, February 2020. doi: 10.1038/s41592-019-0686-2

[Winn11] M. D. Winn, C. C. Ballard, K. D. Cowtan, E. J. Dodson, P. Emsley, P. R. Evans, R .M. Keegan, E. B. Krissnel, A. G. W. Leslie, A. McCoy, S. J. McNicholas, G .N. Murshudov, N. S. Pannu, E. A. Potteron, H .R. Powell, R. J. Read, A. Vagin, and K. S. Wilson. *Overview of the CCP4 suite and current developments*, Acta Cryst., D67(4):235-242, April 2011. doi: 10.1107/S0907444910045749

signac: Data Management and Workflows for Computational Researchers

Bradley D. Dice[‡†*], Brandon L. Butler[§†*], Vyas Ramasubramani[§], Alyssa Travitz[¶], Michael M. Henry[‖], Hardik Ojha[**], Kelly L. Wang[¶], Carl S. Adorf[§], Eric Jankowski[‖], Sharon C. Glotzer[‡§¶††]

Abstract—The **signac** data management framework (https://signac.io) helps researchers execute reproducible computational studies, scales workflows from laptops to supercomputers, and emphasizes portability and fast prototyping. With **signac**, users can track, search, and archive data and metadata for file-based workflows and automate workflow submission on high performance computing (HPC) clusters. We will discuss recent improvements to the software's feature set, scalability, scientific applications, usability, and community. Newly implemented synced data structures, features for generalized workflow execution, and performance optimizations will be covered, as well as recent research using the framework and changes to the project's outreach and governance as a response to its growth.

Index Terms—data management, data science, database, simulation, collaboration, workflow, HPC, reproducibility

Introduction

Scientific research addresses problems where questions often change rapidly, data models are always in flux, and compute infrastructure varies widely from project to project. The **signac** data management framework [ADRG18] is a tool designed by researchers, for researchers, to simplify the process of prototyping and then performing reproducible scientific computations. It forgoes encoding complex data files into a database in favor of working directly on file systems, providing fast indexing utilities for a set of directories. Using **signac**, a data space on the file system can be initialized, searched, and modified using either a Python or command-line interface. By its general-purpose design, **signac** is agnostic to data content and format. The companion package **signac-flow** interacts with the data space to generate and analyze data through reproducible workflows that scale from laptops to supercomputers. Arbitrary shell commands can be run by **signac-flow** as part of a workflow, making it as flexible as a script in any language of choice.

† *These authors contributed equally.*
* *Corresponding author: bdice@umich.edu, butlerbr@umich.edu*
‡ *Department of Physics, University of Michigan, Ann Arbor*
* *Corresponding author: bdice@umich.edu, butlerbr@umich.edu*
§ *Department of Chemical Engineering, University of Michigan, Ann Arbor*
¶ *Macromolecular Science and Engineering Program, University of Michigan, Ann Arbor*
‖ *Micron School of Materials Science and Engineering, Boise State University*
** *Department of Chemical Engineering, Indian Institute of Technology Roorkee*
†† *Biointerfaces Institute, University of Michigan, Ann Arbor*

This paper will focus on developments to the **signac** framework over the last 3 years, during which features, flexibility, usability, and performance have been greatly improved. The core data structures in **signac** have been overhauled to provide a powerful and generic implementation of *synced collections*, that we will leverage in future versions of **signac** to enable more performant data indexing and flexible data layouts. In **signac-flow**, we have added support for submitting *groups* of operations with conditional dependencies, allowing for more efficient utilization of large HPC resources. Further developments allow for operations to act on arbitrary subsets of the data space via *aggregation*, rather than single jobs alone. Moving beyond code development, this paper will also discuss the scientific research these features have enabled and organizational developments supported through key partnerships. We will share our project's experience in continuously revising project governance to encourage sustained contributions, adding more entry points for learning about the project (Slack support, weekly public office hours), and participating in Google Summer of Code in 2020 as a NumFOCUS Affiliated Project. Much of the work has been carried out in conjunction with the Molecular Simulation Design Framework (MoSDeF) [CMI+21], a National Science Foundation Cyberinfrastructure for Sustained Scientific Innovation (CSSI) effort.

Structure and implementation

With **signac**, file-based data and metadata are organized in folders and JSON files, respectively (see Figure 1). A **signac** data space, or *workspace*, contains jobs, which are individual directories associated with a single primary key known as a *state point* stored in a file `signac_statepoint.json` in that directory. The JSON files allow **signac** to index the data space, providing a database-like interface to a collection of directories. Arbitrary user data may be stored in user-created files in these jobs, although **signac** also provides convenient facilities for storing simple lightweight data or array-like data via JSON (the "job document") and HDF5 (the "job data") utilities. Readers seeking more details about **signac** are referred to past **signac** papers [ADRG18], [RAD+18] as well as the **signac** website[1] and documentation[2].

This filesystem-based approach has both advantages and disadvantages. Its key advantages lie in flexibility and portability. The serverless design removes the need for any external running server process, making it easy to operate on any filesystem. The

1. https://signac.io
2. https://docs.signac.io

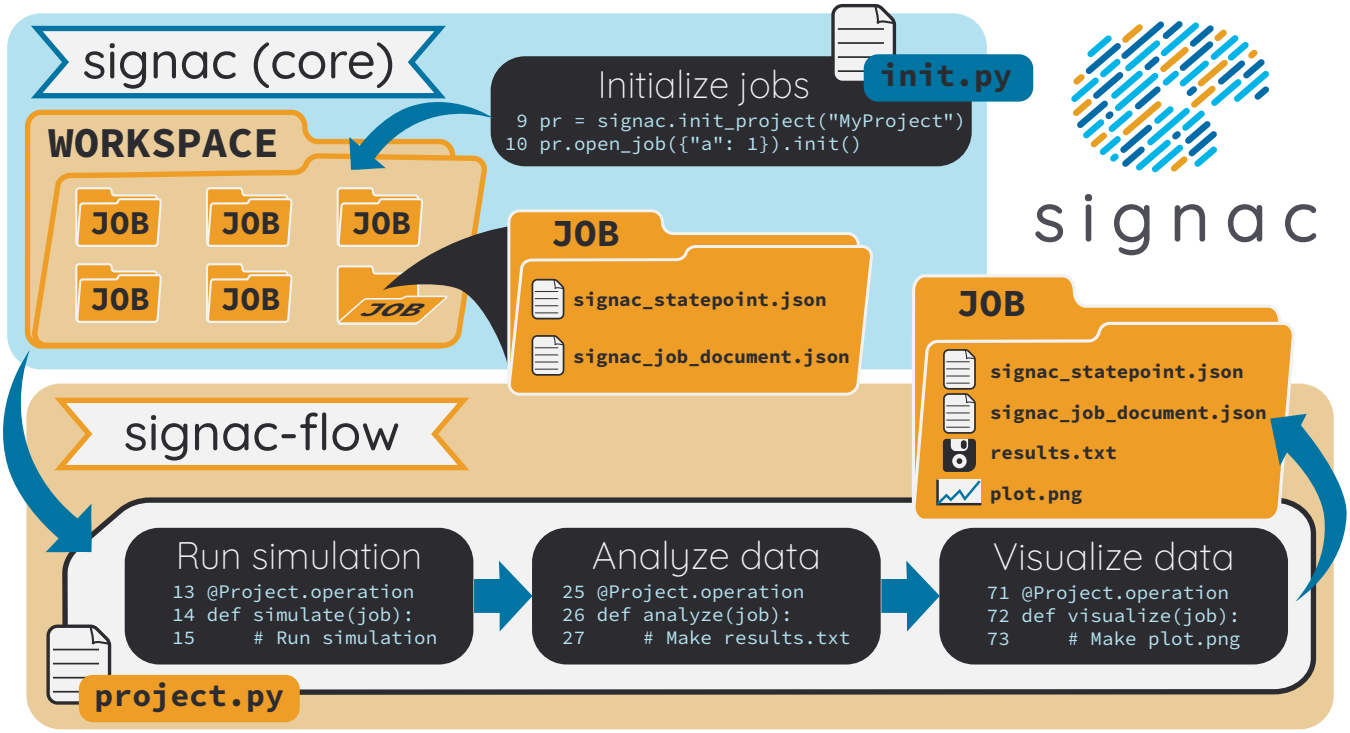

Fig. 1: *Overview of the* ***signac*** *framework. Users first create a project, which initializes a workspace directory on disk. Users define state points which are dictionaries that uniquely identify a job. The workspace holds a directory for each job, containing JSON files that store the state point and job document. The job directory name is a hash of the state point's contents. Here, the* `init.py` *file initializes an empty project and adds one job with state point* `{"a": 1}`*. Next, users define a workflow using a subclass of* ***signac-flow****'s* `FlowProject`*. The workflow shown has three operations (simulate, analyze, visualize) that, when executed, produce two new files* `results.txt` *and* `plot.png` *in the job directory.*

design is also intrinsically distributed, making it well suited for highly parallel workflows where multiple processes concurrently read or write file-based data stored in job directories. Conversely, this distributed approach precludes the performance advantages of centralized data stores with persistent indexes in memory. Typically, the **signac** approach works very well for projects up to 100,000 jobs, while significantly larger projects may have wait times that constrain interactive usage. These limits are inherent to **signac**'s use of small files for each job's state point, but the framework has been aggressively optimized and uses extensive caching/buffering to maximize the achievable throughput within this model.

The framework is a strong choice for applications meeting one or more of the following criteria:

- input/output data is primarily file-based
- prototype research code where data schemas may change or evolve
- computations will use an HPC cluster
- the amount of computation per job is large
- parameter sweeps over a range of values (with values on a grid or dynamically determined by e.g. active learning)
- heterogeneous data (not all jobs have the same keys present in their state points)

For example, M. W. Thompson *et al.* in [TMS+] used 396 jobs/state points to execute computer simulations of room-temperature ionic liquids with GROMACS [PPS+], [LHvdS], [HKvdSL], [AMS+] simulations. The study investigated 18 compositions (by mass fraction) and 22 unique solvents from five chemical families (nitriles, alcohols, halocarbons, carbonyls, and glymes), with a state point for each pairing of mass fraction and solvent type.

Users working with large tabular data (e.g. flat files on disk or data from a SQL database) may prefer to use libraries like pandas [pdt20], [McK], Dask [Tea16], [Roc15], or RAPIDS [Tea18] that are specifically designed for those use cases. However, it is possible to create a **signac** project with state points corresponding to each row, which may be a good use of **signac** if there is file-based data affiliated with each row's parameters.

Code examples of features presented in this paper can be found online[3].

Applications of signac

The **signac** framework has been cited 54 times, according to Google Scholar, and has been used in a range of scientific fields with various types of computational workflows. Some of these studies include quantum calculations of small molecules [GG18], 4,480 simulations of epoxy curing (each containing millions of particles) [TAH+18], inverse design of pair potentials [AADG18], identifying photonic band gaps in 151,593 crystal structures [CADG21], benchmarking atom-density representations for use in machine learning [MVG+21], simulating fluid flow in polymer solutions [PHMTN19], design of optical metamaterials [HCVM20], and economic analysis of drought

3. https://github.com/glotzerlab/signac-examples

risk in agriculture [RD20]. To date, **signac** users have built workflows utilizing a wide range of software packages including simulation tools such as Cassandra and MoSDeF-Cassandra [SMRM+17], [DMD+21], foyer [KST+], GROMACS [PPS+], [LHvdS], [HKvdSL], [AMS+], HOOMD-blue [AGG], [GNA+], [BLBVRJAASCG20], mBuild [KSJ+], MIT Photonic Bands [JJ01], Quantum-ESPRESSO [GBB+09], Rigorous Coupled Wave Analysis (RCWA) [LF12], and VASP [KF96], machine learning libraries including Keras [C+15], scikit-learn [PVG+11], and TensorFlow [AAB+15], and analysis libraries for postprocessing data such as freud [RDH+20], librascal [MVG+21], MDAnalysis [MADWB11], MDTraj [MBH+15], and OVITO [Stu]. Much of the published research using **signac** comes from chemical engineering, materials science, or physics, the fields of many of **signac**'s core developers and thus fields where the project has had greatest exposure. Computational materials research commonly requires large HPC resources with shared file systems, a use case where **signac** excels. However, there are many other fields with similar hardware needs where **signac** can be applied. These include simulation-heavy HPC workloads such as fluid dynamics, atomic/nuclear physics, or genomics, data-intensive fields such as economics or machine learning, and applications needing fast, flexible prototypes for optimization and data analysis.

While there is no "typical" **signac** project, factors such as computational complexity and data sizes offer some rough guidelines for when **signac**'s database-on-the-filesystem is appropriate. For instance, the time to check the status of a workflow depends on the number of jobs, number of operations, and number of conditions to evaluate for those jobs. Typical **signac** projects have 100 to 10,000 jobs, with each job workspace containing arbitrarily large data sizes (the total file size of the job workspace has little effect on the speed of the **signac** framework). To give a rough idea of the limits of scalability, **signac** projects can contain up to around 100,000 jobs while keeping common tasks like checking workflow status in an "interactive" time scale of 1-2 minutes. Some users that primarily wish to leverage **signac-flow**'s workflows for execution and submission may have a very small number of jobs (< 10). One example of this would be executing a small number of expensive biomolecular simulations using different random seeds in each job's state point. Importantly, projects with a small number of jobs can be expanded at a later time, and make use of the same workflow defined for the initial set of jobs. The abilities to grow a project and change its schema on-the-fly catalyze the kind of exploration that is crucial to answering research questions.

The workflow submission features of **signac-flow** interoperates with popular HPC schedulers including SLURM, PBS/TORQUE, and LSF automating the generation and submission of scheduler batch scripts. Directives are set through Python decorators and define resource and execution requests for operations. Examples of directives include number of CPUs or GPUs, the walltime, and memory. The use of directives allows **signac-flow** workflows to be portable across HPC systems by generating resource requests that are specific to each machine's scheduler.

Overview of new features

The last three years of development of the **signac** framework have expanded its usability, feature set, user and developer documentation, and potential applications. Some of the largest architectural changes in the framework will be discussed in their own sections, namely extensions of the workflow model (support for executing groups of operations and aggregators that allow operations to act on multiple jobs) and a much more performant and flexible re-implementation of the core "data structure" classes that synchronize **signac**'s Python representation of state points and job documents with JSON-encoded dictionaries on disk.

Data archival

The primary purpose of the core **signac** package is to simplify and accelerate data management. The **signac** command line interface is a common entry point for users, and provides subcommands for searching, reading, and modifying the data space. New commands for import and export simplify the process of archiving **signac** projects into a structure that is both human-readable and machine-readable for future access (with or without **signac**). Archival is an integral part of research data operations that is frequently overlooked. By using highly compatible and long-lived formats such as JSON for core data storage with simple name schemes, **signac** aims to preserve projects and make it easier for studies to be independently reproduced. This is aligned with the principles of TRUE (Transparent, Reproducible, Usable by others, and Extensible) simulations put forth by the MoSDeF collaboration [TGM+20].

Improved data storage, retrieval, and integrations

Data access via the shell: The `signac shell` command allows the user to quickly enter a Python interpreter that is pre-populated with variables for the current project or job (when in a project or job directory). This means that manipulating a job document or reading data can be done through a hybrid of bash/shell commands and Python commands that are fast to type.

```
~/project $ ls
signac.rc workspace
~/project $ cd workspace/42b7b4f2921788e.../
~/project/workspace/42b7b4f2921788e... $ signac shell
Python 3.8.3
signac 1.6.0

Project:        test
Job:            42b7b4f2921788ea14dac5566e6f06d0
Root:           ~/project
Workspace:      ~/project/workspace
Size:           1

Interact with the project interface using the
"project" or "pr" variable. Type "help(project)"
or "help(signac)" for more information.

>>> job.sp
{'a': 1}
```

HDF5 support for storing numerical data: Many applications used in research generate or consume large numerical arrays. For applications in Python, NumPy arrays are a de facto standard for in-memory representation and manipulation. However, saving these arrays to disk and handling data structures that mix dictionaries and numerical arrays can be cumbersome. The **signac** H5Store feature offers users a convenient wrapper around the h5py library [Col13] for loading and saving both hierarchical/key-value data and numerical array data in the widely-used HDF5 format [Gro21]. The `job.data` attribute is an instance of the `H5Store` class, and is a key-value store saved on disk as `signac_data.h5` in the job workspace. Users who prefer to split data across multiple files can use the `job.stores` API to save in multiple HDF5 files. Corresponding `project.data` and

`project.stores` attributes exist, which save data files in the project root directory. Using an instance of `H5Store` as a context manager allows users to keep the HDF5 file open while reading large chunks of the data:

```
with job.data:
    # Copy array data from the file to memory
    # (which will persist after the HDF5 file is
    # closed) by indexing with an empty tuple:
    my_array = job.data["my_array"][()]
```

Advanced searching and filtering of the workspace: The `signac diff` command, available on both the command line and Python interfaces, returns the difference between two or more state points and allows for easily assessing subsets of the data space. By unifying state point and document queries, filtering, and searching workspaces can be more fine-grained and intuitive.

Data visualization and integrations

Integrating with the PyData ecosystem: Users can now summarize data from a **signac** project into a pandas DataFrame for analysis. The `project.to_dataframe()` feature exports state point and job document information to a pandas DataFrame in a consistent way that allows for quick analysis of all jobs' data. Support for Jupyter notebooks [KRKP+16] has also been added, enabling rich HTML representations of **signac** objects.

Dashboards: The companion package **signac-dashboard** allows users to quickly visualize data stored in a **signac** data space. The dashboard runs in a browser and allows users to display job state points, edit job documents, render images and videos, download any file from a job workspace, and search or browse through state points in their project. Dashboards can be hosted on remote servers and accessed via port forwarding, which makes it possible to review data generated on a remote HPC system without needing to copy it back to a local system for inspection. Users can quickly save notes into the job document and then search those notes, which is useful for high throughput studies that require some manual investigation (e.g. reviewing plots).

Performance enhancements

In early 2021, a significant portion of the codebase was profiled and refactored to improve performance and these improvements were released in **signac** 1.6.0 and **signac-flow** 0.12.0. As a result of these changes, large **signac** projects saw 4-7x speedups for operations such as iterating over the jobs in a project compared to the 1.5.0 release of **signac**. Similarly, performance of a sample workflow that checks status, runs, and submits a FlowProject with 1,000 jobs, 3 operations, and 2 label functions improved roughly 4x compared to **signac-flow** 0.11.0. These improvements allow **signac** to scale to ~100,000 jobs.

In **signac**, the core of the `Project` and `Job` classes were refactored to support lazy attribute access and delayed initialization, which greatly reduces the total amount of disk I/O by waiting until data is actually requested by the user. Other improvements include early exits in functions, reducing the number of required system calls with smarter usage of the `os` library, and switching to algorithms that operate in constant time, $O(1)$, instead of linear time, $O(N_{jobs})$. Optimizations were identified by profiling the performance of common operations on small and large real-world projects with cProfile and visualized with snakeviz [Dav].

Similarly, performance enhancements were also made in the **signac-flow** package. Some of the optimizations identified include lazy evaluation of run commands and directives, and caching of job status information. In addition, the improvements in **signac** such as faster iteration over large **signac** projects used in **signac-flow** made **signac-flow**'s primary functions — checking project status, executing operations, and submitting operations to a cluster — significantly faster.

Improved user output

Workflow graph detection: The preconditions and postconditions of operations in a **signac-flow** `FlowProject` implicitly define a graph. For example, if the operation "analyze" depends on the operation "simulate" via the precondition `@FlowProject.pre.after(simulate)`, then there is a directed edge from "simulate" to "analyze." This graph can now be detected from the workflow conditions and returned in a NetworkX [HSS08] compatible format for display or inspection.

Templated status output: Querying the status of a **signac-flow** project now has many options controlling the information displayed and has been templated to allow for plain text, Markdown, or HTML output. In doing so, the output has also become cleaner and compatible with external tools.

Enhanced workflows

Directives: Execution directives (or *directives* for short) provide a way to specify required resources on HPC schedulers such as number of CPUs/GPUs, MPI ranks, OpenMP threads, walltime, memory, and others. Directives can be a function of the job as well as the operation, allowing for great flexibility. In addition, directives work seamlessly with operation groups, job aggregation, and submission bundling (all of which are described in the following section).

Dynamic workspaces: The **signac-flow** package can now handle workspaces where jobs are created as the result of operations on other jobs. This is crucial for optimization workflows and iteratively sampling parameter spaces, and allows projects to become more automated with some data points only run if a prior condition on another data point is reached.

Executing complex workflows via groups and aggregation

Two new concepts in **signac-flow** provide users with significantly more power to implement complex workflows: *groups* and *aggregation*. A related third concept – *bundling* – which is not new, also provides flexibility to users in their workflows, but exclusively affects scheduler submission, not workflow definition. Figure 2 show a graphical illustration of the three concepts.

As the names of both groups and aggregation imply, the features enable the "grouping" or "aggregating" of existing concepts: operations in the case of groups and jobs in the case of aggregates. The conceptual model of **signac-flow** builds on **signac**'s notions of the `Project` and `Job` (the unit of the data space) through a `FlowProject` class that adds the ability to define and execute operations (the unit of a workflow) that act on jobs. Operations are Python functions or shell commands that act on a job within the data space, and are defined using Python decorator syntax.

```
# project.py
from flow import FlowProject

@FlowProject.operation
@Flowproject.post.true("initialized")
def initialize(job):
    # perform necessary initialize steps
    # for simulation
    job.doc.initialized == True
```

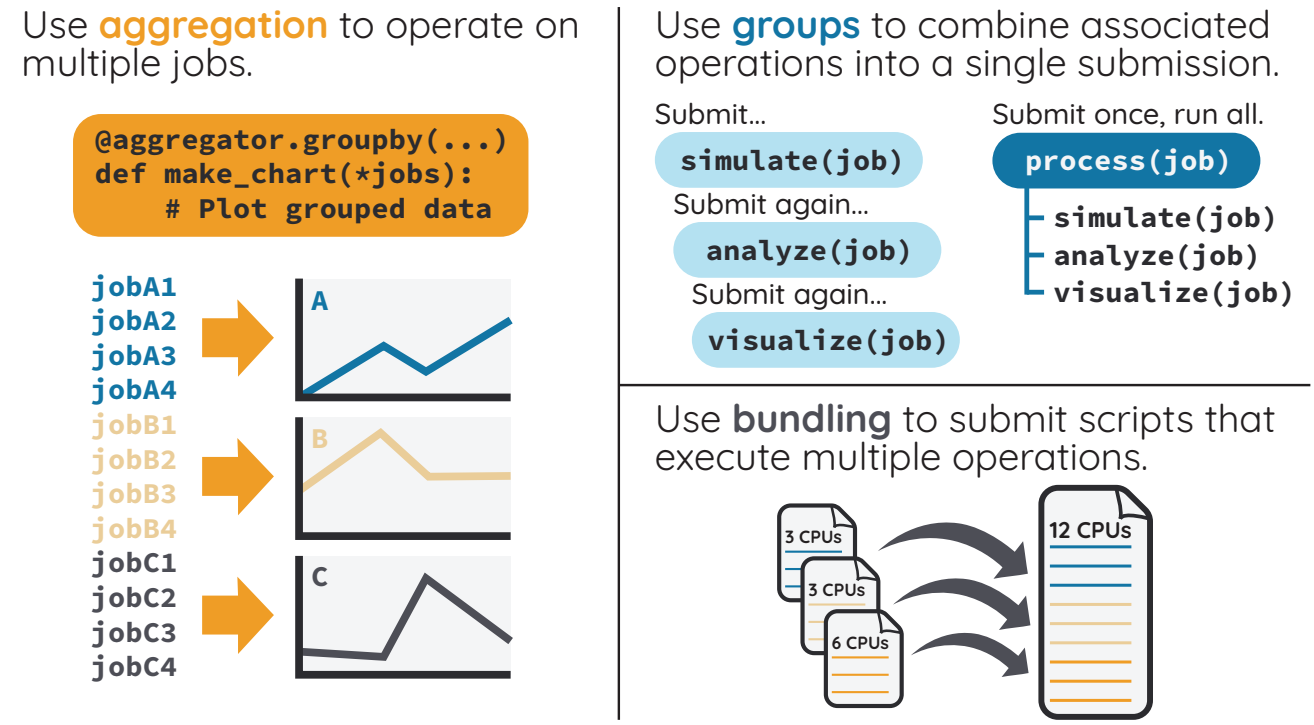

Fig. 2: *Aggregation, groups, and bundling allow users to build complex workflows. The features are orthogonal, and can be used in any combination. Aggregation enables one operation or group to act on multiple jobs. Groups allow users to combine multiple operations into one, with dependencies among operations resolved at run time. Bundling helps users efficiently leverage HPC schedulers by submitting multiple commands in the same script, to be executed in serial or parallel.*

```
if __name__ == "__main__":
    FlowProject().main()
```

When this project is run using **signac-flow**'s command line API (`python project.py run`), the current state point is prepared for simulation. Operations can have preconditions and postconditions that define their eligibility. All preconditions must be met in order for a operation to be eligible for a given job. If all postconditions are met, that indicates an operation is complete (and thus ineligible). Examples of such conditions include the existence of an input file in a job's workspace or a key in the job document (as shown in the above snippet). However, this type of conditional workflow can be inefficient when sequential workflows are coupled with an HPC scheduler interface, because the user must log on to the HPC and submit the next operation after the previous operation is complete. The desire to submit large and long-running jobs to HPC schedulers encourages users to write large operation functions which are not modular and do not accurately represent the individual units of the workflow, thereby limiting **signac-flow**'s utility and reducing the readability of the workflow.

Groups

Groups, implemented by the `FlowGroup` class and `FlowProject.make_group` method, allows users to combine multiple operations into a single entity that can be run or submitted. Submitting a group allows **signac-flow** to dynamically resolve preconditions and postconditions of operations as each operation is executed, making it possible to combine separate operations (e.g. for simulation and analysis and plotting) into a single submission script that will execute eligible operations in sequence. This allows users to write smaller, modular functions, which may require a specific order of execution, without sacrificing the ability to submit large, long-running jobs on HPCs. Furthermore, groups are aware of directives and can properly combine the directives of their constituent operations to specify resources and quantities like walltime whether executing in parallel or serial.

```
from flow import FlowProject

example_group = FlowProject.make_group(
    name="example_group")

@example_group.with_directives(
    {"ngpu": 2,
     "walltime": lambda job: job.doc.hours_to_run})
@FlowProject.post.true("simulated")
@FlowProject.operation
def simulate(job):
    # run simulation
    job.doc.simulated = True

@example_group
@FlowProject.pre.after(simulate)
@FlowProject.post.true("analyzed")
@FlowProject.operation
def analyze(job):
    # analyze simulation results
    job.doc.analyzed = True
```

Groups also allow for specifying multiple machine specific resources (CPU or GPU) with the same operation. An operation can have unique directives for each distinct group to which it belongs. By associating an operation's directives with respect to a specific group, groups can represent distinct compute environments, such as a local workstation or a remote supercomputing cluster. The below snippet shows an `expensive_simulate` operation which can be executed with three different directives depending on how

it is written. If executed through `cpu_group` the operation will request 48 cores, if `gpu_group` 4 GPUs, if neither then it will request 4 cores. This represents the real use case where a user may want to run an operation locally (in this case without a group), or on a CPU or GPU focused HPC/workstation.

```
from flow import FlowProject

cpu_group = FlowProject.make_group(name="cpu")
gpu_group = FlowProject.make_group(name="gpu")

@cpu_group.with_directives({"np": 48})
@gpu_group.with_directives({"ngpu": 4})
@FlowProject.operation.with_directives({"np": 4})
def expensive_simulate(job):
    # expensive simulation run on CPUs or GPUs
    pass
```

Aggregation

Users also frequently work with multiple jobs when performing tasks such as plotting data from all jobs in the same figure. Though the **signac** package has methods like `Project.groupby`, which can generate subsets of the project that are grouped by a state point key, there has been no way to use these "aggregation" features in **signac-flow** for defining workflows. The concept of aggregation provides a straightforward way for users to write and submit operations that act on arbitrary subsets of jobs in a **signac** data space through functions analogous to `Project.groupby`. Just as the groups feature acts as an abstraction over operations, aggregation can be viewed as an abstraction over jobs. When decorated with an aggregator, operations can accept multiple job instances as positional arguments through Python's argument unpacking. Decorators are used to define aggregates, encompassed in the `@aggregator` decorator for single operations and in the argument `aggregator_function` to `FlowProject.make_group` for groups of operations.

```
from flow import FlowProject

@aggregator
@FlowProject.operation
def plot_enzyme_activity(*jobs):
    import matplotlib.pyplot as plt
    import numpy as np

    x = [job.sp.temperature for job in jobs]
    y = [job.doc.activity for job in jobs]
    fig, ax = plt.subplots()
    ax.scatter(x, y)
    ax.set_title(
        "Enzymatic Activity Across Temperature")
    fig.savefig("enzyme-activity.png")
```

Like groups, there are many reasons why a user might wish to use aggregation. For example, a **signac** data space that describes weather data for multiple cities in multiple years might want to plot or analyze data that uses `@aggregator.groupby("city")` to show changes over time for each city in the data space. Similarly, aggregating over replicas (e.g. the same simulation with different random seeds) facilitates computing averaged quantities and error bars. Another example is submitting aggregates with a fixed number of jobs in each aggregate to enable massive parallelization by breaking a large MPI communicator into a smaller communicator for each independent job, which is necessary for efficient utilization of leadership-class supercomputers like OLCF Summit.

Bundling

Finally, bundling is another way to use workflows in conjunction with an HPC scheduling system. Whereas aggregates are concerned with jobs and groups operations, bundling is concerned with combining executable units into a single submission script. This distinction means that bundling is not part of the workflow definition, but is a means of tailoring batch scripts for different HPC systems. Bundles allow users to leverage scheduler resources effectively and minimize queue time, and can be run in serial (the default) or parallel. Users enable bundling by passing the command line argument `--bundle`, optionally with another argument `--parallel` to run each command in the bundle in parallel (the Python API has corresponding options as well). The simplest case of a bundle is a submission script with the same operation being executed for multiple jobs. Bundling is what allows the submission script to contain multiple jobs executing the same operation. By storing information about the generated bundles during submission, **signac-flow** prevents accidental resubmission just as in the unbundled case. While the example mentioned above does not use either groups or aggregation, bundles works seamlessly with both.

Cluster templates

The **signac-flow** software includes automatic detection and script support for SLURM, PBS/TORQUE, and LSF schedulers. However, effective HPC utilization frequently relies on specific information such as numbers of cores per compute node or designated partitions for GPU or large memory applications. To this end, **signac-flow** includes templates for a number of HPC clusters including OLCF Summit and Andes, XSEDE [TCD+14] clusters such as PSC Bridges-2, SDSC Comet, and TACC Stampede2, and university clusters such as the University of Michigan's Great Lakes and University of Minnesota's Mangi. These cluster templates change frequently as HPC systems are brought online and later decommissioned. Users can create their own templates to contribute to the package or use locally.

Synced collections: backend-agnostic, persistent, mutable data structures

Motivation

At its core, **signac** is a tool for organizing and working with data on the filesystem, presenting a Pythonic interface for tasks like creating directories and modifying files. In particular, **signac** makes modifying the JSON files used to store a job's state points and documents as easy as working with Python dictionaries. Despite heavy optimization, when seeking to scale **signac** to ever-larger data spaces, we quickly realized that the most significant performance barrier was the overhead of parsing and modifying large numbers of text files. Unfortunately, the usage of JSON files in this manner was deeply embedded in our data model, which made switching to a more performant backend without breaking APIs or severely complicating our data model a daunting task.

While attempting to separate the **signac** data model from its original backend implementation (manipulating JSON files on disk), we identified a common pattern: providing a dictionary-like interface for an underlying resource. Several well-known Python packages such as h5py [Col13] and zarr [MjD+20] also use dictionary-like interfaces to make working with complex resources feel natural to Python users. Most such packages implement this layer directly for their particular use case, but the nature of the

problem suggested to us the possibility of developing a more generic representation of this interface. Indeed, the purpose of the Python standard library's `collections.abc` module to make it easy to define objects that "look like" standard Python objects while having completely customizable behavior under the hood. As such, we saw an opportunity to specialize this pattern for a specific use case: the transparent synchronization of a Python object with an underlying resource.

The *synced collections* framework represents the culmination of our efforts in this direction, providing a generic framework in which interfaces of any abstract data type can be mapped to arbitrary underlying synchronization protocols. In **signac**, this framework allows us to hide the details of a particular file storage medium (like JSON) behind a dictionary-like interface, but it can just as easily be used for tasks such as creating a new, list-like interface that automatically saves all its data in a plain-text CSV format. This section will offer a high-level overview of the synced collections framework and our plans for its use within **signac**, with an eye to potential users in other domains as well.

Summary of features

We designed synced collections to be flexible, easily extensible, and independent of **signac**'s data model. Most practical use cases for this framework involve an underlying resource that may be modified by any number of associated in-memory objects that behave like standard Python collections, such as dictionaries or lists. Therefore, all normal operations must be preceded by loading from this resource and updating the in-memory store, and they must be succeeded by saving to that resource. The central idea behind synced collections is to decouple this process into two distinct groups of tasks: the saving and loading of data from a particular resource backend, and the synchronization of two in-memory objects of a given type. This delineation allows us to, for instance, encapsulate all logic for JSON files into a single `JSONCollection` class and then combine it with dictionary- or list-like `SyncedDict`/`SyncedList` classes via inheritance to create fully functional JSON-backed dictionaries or lists. Such synchronization significantly lowers performance, so the framework also exposes an API to implement buffering protocols to collect operations into a single transaction before submitting them to the underlying resource.

Previously, **signac** contained a single `JSONDict` class as part of its API, along with a separately implemented internal-facing `JSONList` that could only be used as a member of a `JSONDict`. With the new framework, users can create fully-functional, arbitrarily nested `JSONDict` and `JSONList` objects that share the same logic for reading from and writing to JSON files. Just as importantly, **signac** can now combine these data structures with a different backend, allowing us to swap in different storage mechanisms for improved performance and flexibility with no change in our APIs. Since different types of resources may have different approaches to batching transactions — for example, a SQLite backend may want to exploit true SQL transactions, while a Redis backend might simply collect all changes in memory and delay sending memory to the server — synced collections also support customizable buffering protocols, again via class inheritance.

Applications of synced collections

The new synced collections promise to substantially simplify both feature and performance enhancements to the **signac** framework. Performance improvements in the form of Redis-based storage are already possible with synced collections, and as expected they show substantial speedups over the current JSON-based approach. We have also exploited the new and more flexible buffering protocol to implement and test alternatives to the previous approach. In certain cases, our new buffering techniques improve performance of buffered operations by 1-2 orders of magnitude. Some of these performance improvements are drop-in replacements that require no changes to our existing data models, and we plan to enable these in upcoming versions of **signac**.

The generality of synced collections makes them broadly useful even outside the **signac** framework. Adding Pythonic APIs to collection-like objects can be challenging, particularly when those objects should support arbitrary nesting, but synced collections enable nesting as a core feature to dramatically simplify this process. Moreover, while the framework was originally conceived to support synchronization of an in-memory data structure with a resource on disk, it can also be used to synchronize with another in-memory resource. A powerful example of this would be wrapping a C or C++ extension type, for instance by creating a `SyncedList` that synchronizes with a C++ `std::vector`, such that changes to either object would be transparently reflected in the other. With synced collections, creating this class just requires defining a conversion between a `std::vector` and a raw Python list, a trivial task using standard tools for exposing extension types such as pybind or Cython.

At a higher level, synced collections represent an important step in improving both the scalability and flexibility of **signac**. By abstracting away details of persistent file storage from the rest of **signac**, they make it much easier for the rest of **signac** to focus on offering flexible data models. One of the most common use cases of **signac** is creating data spaces with homogeneous schemas that fit naturally into tabular data structures. In future iterations of **signac**, we plan to allow users to opt into homogeneous schemas, which would enable us to replace file-based indexes with SQL-backed databases that would offer orders of magnitude in performance improvements. Using this flexibility, we could also move away from our currently rigid workspace model to allow more general data layouts on disk for cases where users may benefit from more general folder structures. As such, synced collections are a stepping stone to creating a more general and powerful version of **signac**.

Project evolution

The **signac** project has evolved from being an open-source project mostly developed and managed by the Glotzer Group at the University of Michigan, to being supported by over 30 contributors and 8 committers/maintainers on 3 continents and with over 55 citations from academic and government research labs and 12 talks at large scientific, Python, and data science conferences. The growth in involvement with **signac** results from our focus on developing features based on user needs, as well as our efforts to transition **signac** users into **signac** contributors, through many initiatives in the past few years. Through encouraging users to become contributors, we ensure that **signac** addresses real users' needs. Early on, we identified that the framework had the potential to be used by a wide community of researchers and that its philosophy was aligned with other projects in the scientific Python ecosystem. We have expanded **signac**'s contributor base beyond the University of Michigan through research collaborations such

as the MoSDeF CSSI with other universities, sharing the framework at conferences, and through the Google Summer of Code (GSoC) program, which we applied to under the NumFOCUS organization. Working with and mentoring students through GSoC led to a new committer and significant work on the synced collections and aggregation projects presented above. We provide active support and open discussion for the contributor and user community through Slack. In addition, we have started hosting weekly "office hours" for in-person (virtual) introduction and guided contributions to the code base. By pairing new contributors with experienced **signac** developers, we significantly reduce the knowledge barrier to joining a new project. Close interactions between developers and users during office hours has led to more features and documentation born directly out of user need. Contributing to documentation has been a productive starting point for new users-turned-contributors, both for the users and the project, since it improves the users' familiarity with the API as well as addresses weak spots in the documentation that are more obvious to new users.

In our growth with increasing contributors and users, we recognized a need to change our governance structure to make contributing easier and provide a clear organizational structure to the community. We based our new model on the Meritocratic Governance Model and our manager roles on Numba [LPS] Czars. We decided on a four category system with maintainers, committers, contributors, and users. Code review and pull request merge responsibilities are granted to maintainers and committers, who are (self-) nominated and accepted by a vote of the project maintainers. Maintainers are additionally responsible for the strategic direction of the project and administrative duties. Contributors consist of all members of the community who have contributed in some way to the framework, which includes adding or refactoring code as well as filing issues and improving documentation. Finally, users refer to all those who use **signac** in any capacity.

In addition, to avoid overloading our committers and maintainers, we added three rotating manager roles to our governance model that ensure project management goes smoothly: triage, community, and release. These managers have specific rotation policies based on time (or release cycles). The triage manager role rotates weekly and looks at new issues or pull requests and handles cleanup of outdated issues. The community manager role rotates monthly and is in charge of meeting planning and outreach. Lastly, the release manager rotates with each release cycle and is the primary decision maker for the timeline and feature scope of package releases. This prevents burnout among our senior developers and provides a sense of ownership to a greater number of people, instead of relying on a "benevolent dictator/oligarchy for life" mode of project leadership.

Conclusions

From the birth of the **signac** framework in 2015 to now, **signac** has grown in usability, performance, and use. In the last three years, we have added exciting new features such as groups, aggregation, and synced collections, while learning how to manage outreach and establish sustainable project governance in a burgeoning scientific open-source project. We hope to continue expanding the framework through user-oriented development, reach users in research fields beyond materials science that routinely have projects suited for **signac**, and welcome new contributors with diverse backgrounds and skills to the project.

Installing signac

The **signac** framework is tested for Python 3.6+ and is compatible with Linux, macOS, and Windows. The software is available under the BSD-3 Clause license. To install, execute

```
conda install -c conda-forge signac \
signac-flow signac-dashboard
```

or

```
pip install signac signac-flow signac-dashboard
```

Source code is available on GitHub[4][5] and documentation is hosted online by ReadTheDocs[6].

Acknowledgments

We would also like to thank NumFOCUS for providing helpful advice on open-source governance, project sustainability, and community outreach, as well as funding for the design of the **signac** project logo.

This work was supported by the National Science Foundation, Office of Advanced Cyberinfrastructure Awards OAC 1835612 and OAC 1835593. B.D. and B.B. acknowledge fellowship support from the National Science Foundation under ACI 1547580, S212: Impl: The Molecular Sciences Software Institute [WDC18], [KWB+18]. B.D. was also supported by a National Science Foundation Graduate Research Fellowship Grant DGE 1256260 (2016–2019). V.R. acknowledges the 2019-2020 J. Robert Beyster Computational Innovation Graduate Fellowship at the University of Michigan. A.T. is supported by the National Science Foundation under DMR 1707640. Software was deployed and validated and benchmarked on the Extreme Science and Engineering Discovery Environment (XSEDE) [TCD+14], which is supported by National Science Foundation Grant No. ACI-1053575 (XSEDE award DMR 140129) and on resources of the Oak Ridge Leadership Computing Facility which is a DOE Office of Science User Facility supported under Contract No. DE-AC05-00OR22725.

Author contributions

Conceptualization, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; data curation, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; funding acquisition, E.J. and S.C.G.; methodology, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; project administration, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; software, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; supervision, S.C.G.; visualization, B.D.D., B.L.B., A.T., and K.W.; writing – original draft, B.D.D., B.L.B., V.R., A.T., and H.O.; writing – review & editing, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., K.W., C.S.A., and S.C.G. All authors have read and agreed to the published version of the manuscript.

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

4. https://github.com/glotzerlab/signac
5. https://github.com/glotzerlab/signac-flow
6. https://docs.signac.io/

Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

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

[ADRG18] Carl S. Adorf, Paul M. Dodd, Vyas Ramasubramani, and Sharon C. Glotzer. Simple data and workflow management with the signac framework. *Comput. Mater. Sci.*, 146(C):220–229, 2018. doi:10.1016/j.commatsci.2018.01.035.

[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. doi:10.1016/j.commatsci.2019.109363.

[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. doi:10.1016/j.softx.2015.06.001.

[BLBVRJAASCG20] Brandon L. Butler, Vyas Ramasubramani, Joshua A. Anderson, and Sharon C. Glotzer. HOOMD-blue version 3.0 A Modern, Extensible, Flexible, Object-Oriented API for Molecular Simulations. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 24–31, 2020. doi:10.25080/Majora-342d178e-004.

[C+15] François Chollet et al. Keras, 2015. URL: https://keras.io.

[CADG21] Rose K. Cersonsky, James Antonaglia, Bradley D. Dice, and Sharon C. Glotzer. The diversity of three-dimensional photonic crystals. *Nature Communications*, 12(1):2543, May 2021. doi:10.1038/s41467-021-22809-6.

[CMI+21] Peter T. Cummings, Clare McCabe, Christopher R. Iacovella, Akos Ledeczi, Eric Jankowski, Arthi Jayaraman, Jeremy C. Palmer, Edward J. Maginn, Sharon C. Glotzer, Joshua A. Anderson, J. Ilja Siepmann, Jeffrey Potoff, Ray A. Matsumoto, Justin B. Gilmer, Ryan S. DeFever, Ramanish Singh, and Brad Crawford. Open-source molecular modeling software in chemical engineering focusing on the Molecular Simulation Design Framework. *AIChE Journal*, 67(3):e17206, 2021. doi:10.1002/aic.17206.

[Col13] Andrew Collette. *Python and HDF5*. O'Reilly, 2013.

[Dav] Matt Davis. snakeviz. URL: https://jiffyclub.github.io/snakeviz/.

[DMD+21] Ryan S DeFever, Ray A Matsumoto, Alexander W Dowling, Peter T Cummings, and Edward J Maginn. Mosdef cassandra: A complete python interface for the cassandra monte carlo software. *Journal of Computational Chemistry*, 2021. doi:10.1002/jcc.24807.

[GBB+09] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, sep 2009. doi:10.1088/0953-8984/21/39/395502.

[GG18] Marco Govoni and Giulia Galli. Gw100: Comparison of methods and accuracy of results obtained with the west code. *Journal of Chemical Theory and Computation*, 14(4):1895–1909, 2018. doi:10.1021/acs.jctc.7b00952.

[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. doi:10.1016/j.cpc.2015.02.028.

[Gro21] The HDF Group. Hierarchical data format, version 5, 1997-2021. URL: https://www.hdfgroup.org/HDF5/.

[HCVM20] Eric S. Harper, Eleanor J. Coyle, Jonathan P. Vernon, and Matthew S. Mills. Inverse design of broadband highly reflective metasurfaces using neural networks. *Physical Review B*, 101(19):195104, May 2020. doi:10.1103/PhysRevB.101.195104.

[HKvdSL] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. 4(3):435–447. doi:10.1021/ct700301q.

[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.

[JJ01] Steven G. Johnson and J. D. Joannopoulos. Block-iterative frequency-domain methods for maxwell's equations in a planewave basis. *Opt. Express*, 8(3):173–190, Jan 2001. doi:10.1364/OE.8.000173.

[KF96] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996. doi:10.1103/PhysRevB.54.11169.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[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. 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. doi:10.1016/j.commatsci.2019.05.026.

[KWB+18] Anna Krylov, Theresa L. Windus, Taylor Barnes, Eliseo Marin-Rimoldi, Jessica A. Nash, Benjamin Pritchard, Daniel G. A. Smith, Doaa Altarawy, Paul Saxe, Cecilia Clementi, T. Daniel Crawford, Robert J. Harrison, Shantenu Jha, Vijay S. Pande, and Teresa Head-Gordon. Perspective: Computational chemistry software and its advancement as illustrated through three grand challenge cases for molecular science. *The Journal of Chemical Physics*, 149(18):180901, 2018. doi:10.1063/1.5052551.

[LF12] Victor Liu and Shanhui Fan. S4 : A free electromagnetic solver for layered periodic structures. *Computer Physics Communications*, 183(10):2233–2244, 2012. doi:10.1016/j.cpc.2012.04.026.

[LHvdS] Erik Lindahl, Berk Hess, and David van der Spoel.

GROMACS 3.0: A package for molecular simulation and trajectory analysis. 7(8):306–317. doi:10.1007/s008940100045.

[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. doi:10.1145/2833157.2833162.

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

[MBH+15] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. Mdtraj: A modern open library for the analysis of molecular dynamics trajectories. *Biophysical Journal*, 109(8):1528–1532, 2015. doi:10.1016/j.bpj.2015.08.015.

[McK] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61. doi:10.25080/Majora-92bf1922-00a.

[MjD+20] Alistair Miles, jakirkham, Martin Durant, Matthias Bussonnier, James Bourbeau, Tarik Onalan, Joe Hamman, Zain Patel, Matthew Rocklin, shikharsg, Ryan Abernathey, Josh Moore, Vincent Schut, raphael dussin, Elliott Sales de Andrade, Charles Noyes, Aleksandar Jelenak, Anderson Banihirwe, Chris Barnes, George Sakkis, Jan Funke, Jerome Kelleher, Joe Jevnik, Justin Swaney, Poruri Sai Rahul, Stephan Saalfeld, john, Tommy Tran, pyup.io bot, and sbalmer. zarr-developers/zarr-python: v2.5.0, October 2020. doi:10.5281/zenodo.4069231.

[MVG+21] Félix Musil, Max Veit, Alexander Goscinski, Guillaume Fraux, Michael J. Willatt, Markus Stricker, Till Junge, and Michele Ceriotti. Efficient implementation of atom-density representations. *The Journal of Chemical Physics*, 154(11):114109, March 2021. doi:10.1063/5.0044689.

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

[PHMTN19] Michael P. Howard, Thomas M. Truskett, and Arash Nikoubashman. Cross-stream migration of a Brownian droplet in a polymer solution under Poiseuille flow. *Soft Matter*, 15(15):3168–3178, 2019. doi:10.1039/C8SM02552E.

[PPS+] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GROMACS 4.5: A high-throughput and highly parallel open source molecular simulation toolkit. 29(7):845–854. doi:10.1093/bioinformatics/btt055.

[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.

[RAD+18] Vyas Ramasubramani, Carl S. Adorf, Paul M. Dodd, Bradley D. Dice, and Sharon C. Glotzer. signac: A python framework for data and workflow management. pages 152–159, 2018. doi:10.25080/Majora-4af1f417-016.

[RD20] David Rodziewicz and Jacob Dice. Drought Risk to the Agriculture Sector. *The Federal Reserve Bank of Kansas City Economic Review*, December 2020. doi:10.18651/ER/v105n2RodziewiczDice.

[RDH+20] 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. *Computer Physics Communications*, 254:107275, 2020. doi:10.1016/j.cpc.2020.107275.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130–136, 2015. doi:10.25080/Majora-7b98e3ed-013.

[SMRM+17] 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, et al. Cassandra: An open source monte carlo package for molecular simulation, 2017. doi:10.1002/jcc.26544.

[Stu] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. 18(1):015012. doi:10.1088/0965-0393/18/1/015012.

[TAH+18] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, April 2018. doi:10.1142/S0219633618400059.

[TCD+14] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74, 2014. doi:10.1109/MCSE.2014.80.

[Tea16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[Tea18] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. URL: https://rapids.ai.

[TGM+20] 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). *Molecular Physics*, 118(9-10):e1742938, June 2020. doi:10.1080/00268976.2020.1742938.

[TMS+] Matthew W. Thompson, Ray Matsumoto, Robert L. Sacci, Nicolette C. Sanders, and Peter T. Cummings. Scalable Screening of Soft Matter: A Case Study of Mixtures of Ionic Liquids and Organic Solvents. 123(6):1340–1347. doi:10.1021/acs.jpcb.8b11527.

[WDC18] Nancy Wilkins-Diehr and T. Daniel Crawford. Nsf's inaugural software institutes: The science gateways community institute and the molecular sciences software institute. *Computing in Science Engineering*, 20(5):26–38, 2018. doi:10.1109/MCSE.2018.05329813.

Accelerating Spectroscopic Data Processing Using Python and GPUs on NERSC Supercomputers

Daniel Margala[‡*], Laurie Stephey[‡], Rollin Thomas[‡], Stephen Bailey[§]

Abstract—The Dark Energy Spectroscopic Instrument (DESI) will create the most detailed 3D map of the Universe to date by measuring redshifts in light spectra of over 30 million galaxies. The extraction of 1D spectra from 2D spectrograph traces in the instrument output is one of the main computational bottlenecks of DESI data processing pipeline, which is predominantly implemented in Python. The new Perlmutter supercomputer system at the National Energy Scientific Research and Computing Center (NERSC) will feature over 6,000 NVIDIA A100 GPUs across 1,500 nodes. The new heterogenous CPU-GPU computing capability at NERSC opens the door for improved performance for science applications that are able to leverage the high-throughput computation enabled by GPUs. We have ported the DESI spectral extraction code to run on GPU devices to achieve a 20x improvement in per-node throughput compared to the current state of the art on the CPU-only Haswell partition of the Cori supercomputer system at NERSC.

Index Terms—Python, HPC, GPU, CUDA, MPI, CuPy, Numba, mpi4py, NumPy, SciPy, Astronomy, Spectroscopy

Introduction

The Dark Energy Spectroscopic Instrument (DESI) experiment is a cosmological redshift survey. The survey will create the most detailed 3D map of the Universe to date, using position and redshift information from over 30 million galaxies. During operation, around 1000 CCD frames per night (30 per exposure) are read out from the instrument and transferred to NERSC for processing and analysis. Each frame contains 500 2D spectrograph traces from galaxies, standard stars (for calibration), or just the sky (for background subtraction). These traces must be extracted from the CCD frames taking into account optical effects from the instrument, telescope, and the Earth's atmosphere. Redshifts are measured from the extracted The data is processed in near-real time in order to monitor survey progress and update the observing schedule for the following night. Periodically, a complete reprocessing of all data observed to-date is performed and made available as data release to the collaboration and eventually released to the public.

The DESI spectral extraction code is an implementation of the *spectro-perfectionism* algorithm, described in [BS10]. The process of extracting 1D spectra from 2D spectrograph traces for all 500 targets per frame is computationally intensive and has been the primary focus of optimization efforts for several years ([RTD+17], [STB19]). The DESI data processing pipeline is predominantly implemented using the Python programming language. A strict requirement from the DESI data processing team is to keep the code implementation in Python.

The existing state of the art implementation utilizes a divide and conquer framework to make *spectro-perfectionism* algorithm tractable on existing computing hardware, see Figure 1. The code utilizes the Message Passing Interface (MPI) via *mpi4py* to exploit both multi-core and multi-node parallelism ([DPS05]). The application uses multidimensional array data structures provided by NumPy along with several linear algebra and special functions from the NumPy and SciPy libraries ([HMvdW+20], [VGO+20]). Several expensive kernels are implemented using Numba just-in-time compilation ([LPS15]). All input and output files are stored on disk using the FITS file format. The application is parallelized by dividing an image into thousands of small patches, performing the extraction on each individual patch in parallel, and stitching the result back together.

This has worked well for CPU-only computing architectures such as the Haswell (Intel Xeon Processor E5-2698 v3) and Knights Landing (Intel Xeon Phi Processor 7250) partitions on the Cori[1] supercomputer at NERSC. The new Perlmutter[2] supercomputer system at NERSC will have a partition of GPU accelerated nodes (AMD EPYC 7763, NVIDIA A100 GPU). The goal of this work is to speed up the DESI experiment's data processing pipeline by porting the spectroscopic extraction step to run on the GPU partition of the Perlmutter supercomputer at NERSC.

In early 2020, the team began reimplementing the existing extraction code *specter*[3] by reconsidering the problem. The DESI spectral extraction problem is fundamentally an image processing problem which historically have been well-suited to GPUs. However, in many places, the existing CPU version of the code used loops and branching logic rather than vector or matrix-based operations. We performed a significant refactor switching key parts of the analysis to matrix-based operations which would be well suited to massive GPU parallelism. Additionally, the refactor enabled more flexible task partitioning and improved node utilization. From this refactor alone, still running only on the CPU, we obtained 1.6x speedup compared to the original CPU version. From here, we began our GPU implementation.

* *Corresponding author: danielmargala@lbl.gov*
‡ *Lawrence Berkeley National Laboratory: National Energy Scientific Research and Computing Center*
§ *Lawrence Berkeley National Laboratory: Physics Division*

1. https://docs.nersc.gov/systems/cori/
2. https://docs.nersc.gov/systems/perlmutter/

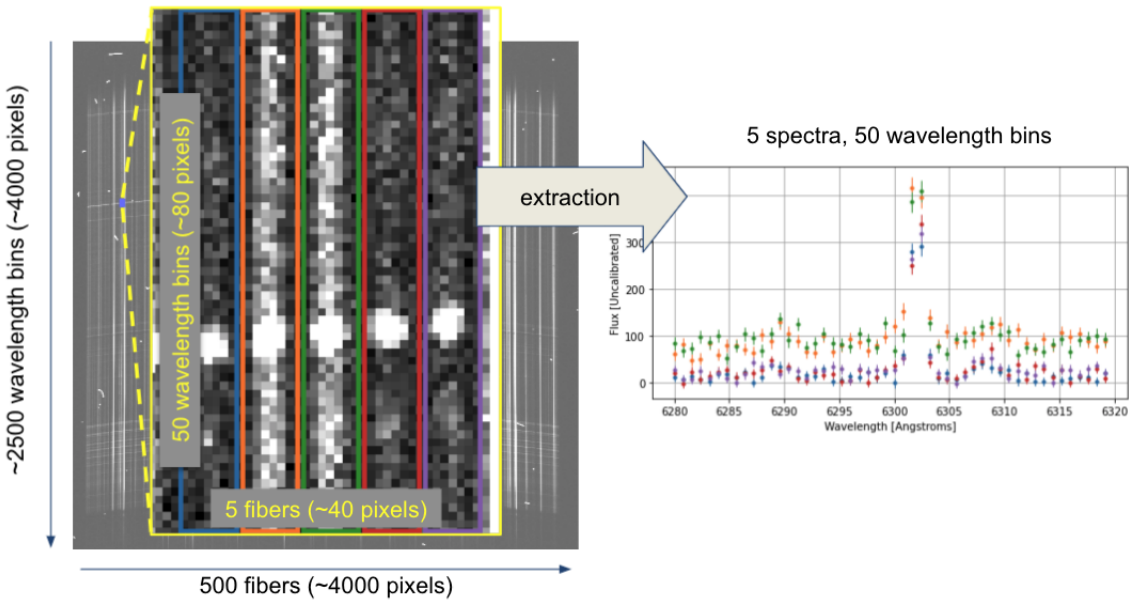

Fig. 1: *The goal of the algorithm is to extract spectra from raw telescope output. Here we show the raw telescope output for a single "patch" and the corresponding pieces of the extracted spectra. The divide and conquer strategy used in this application divides an image into roughly 5,000 patches which can be extracted in parallel. The extracted pieces are then stiched back together and written to disk for further processing by the data pipeline.*

We describe our iterative approach to porting and optimizing the application using NVIDIA Nsight Systems for performance analysis. We use a combination of CuPy and JIT-compiled CUDA kernels via Numba for GPU-acceleration. In order to maximize use of resources (both CPUs and GPUs), we use MPI via mpi4py and CUDA Multi-Process Service. We discuss the lessons we learned during the course of this work that will help guide future efforts of the team and inform other science teams looking to leverage GPU-acceleration in their Python-based data processing applications. We project that new extraction code *gpu_specter*[4] running on Perlmutter will achieve a 20x improvement in per-node throughput compared to the current production throughput on Cori Haswell.

GPU Implementation

The existing CPU implementation uses NumPy and SciPy (BLAS and LAPACK) for linear algebra, numba just-in-time compilation for specialized kernels, and mpi4py (MPI) for multi-core and multi-node scaling. The code is parallelized to run on multiple CPU cores and nodes using a Single Program Multiple Data (SPMD) programming pattern enabled by MPI through mpi4py. The structure of the program is illustrated in Figure 2, which highlights the main MPI communication points.

In order to leverage the compute capabilities of GPU devices and adhere to the DESI Python requirement, we decided to use a GPU-accelerated Python library. The main considerations for heterogeneous CPU-GPU computing are to minimize data movement between the CPU host and the GPU device and to feed the GPU large chunks of data that can be processed in parallel. Keeping those considerations in mind, we left rest of the GPU programming details to external libraries. There are many rapidly maturing Python libraries that allow users to write code that will run on GPU hardware, such as CuPy, pyCUDA, pytorch, JAX, and Numba CUDA. We chose to use CuPy [OUN+17] and Numba CUDA based on our ability to easily integrate their API with our existing code.

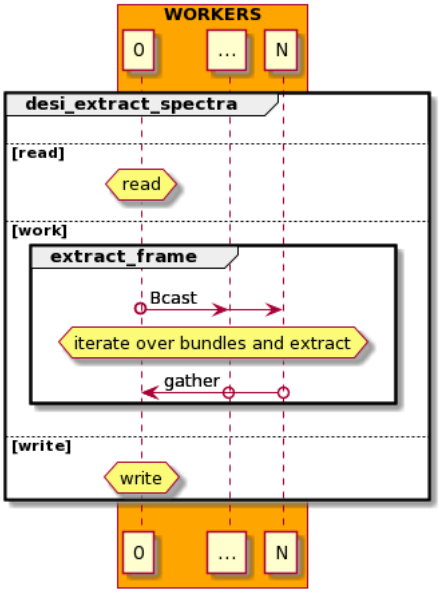

Fig. 2: *An illustration of the program structure highlighting main MPI communication points. Flow runs from top to bottom.*

The initial GPU port was implemented by off-loading compute intensive steps of the extraction to the GPU using CuPy in place of NumPy and SciPy. A few custom kernels were also re-implemented using Numba CUDA just-in-time compilation. In many cases, we merely replaced an existing API call from *numpy*, *scipy*, or *numba.jit* with equivalent GPU-accelerated version from *cupy*, *cupyx.scipy*, or *numba.cuda.jit*.

3. https://github.com/desihub/specter
4. https://github.com/desihub/gpu_specter

The example code below demonstrates how we integrated *cupy*, *numba.cuda*, and the NumPy API:

```
import cupy
import numba.cuda
import numpy

# CUDA kernel
@numba.cuda.jit
def _cuda_addone(x):
    i = numba.cuda.grid(1)
    if i < x.size:
        x[i] += 1

# convenience wrapper with thread/block configuration
def addone(x):
    # threads per block
    tpb = 32
    # blocks per grid
    bpg = (x.size + (tpb - 1)) // tpb
    _cuda_addone[bpg, tpb](x)

# create array on device using cupy
x = cupy.zeros(1000)
# pass cupy ndarray to numba.cuda kernel
addone(x)
# Use numpy api with cupy ndarray
total = numpy.sum(x)
```

We found that this interopability gave us a lot of flexibility to experiment during development. This achieved our initial goal porting the application to run on GPU hardware.

In the following sub-sections, we will discuss the major development milestones that lead to the improved performance of the application on GPUs.

Profiling the Code

As discussed in previous work [STB19], the team found a lot of value using profiling tools such as the *cProfile* Python module. In this work, we used NVIDIA's NSight Systems to profile the application, identify bottlenecks in performance, and focus optimization efforts. We added CUDA NVTX markers (using the CuPy API) to label regions of our code using descriptions that we would be able to easily identify in the profile viewer. Without these labels, it sometimes difficult to decipher the names of low-level kernels that are called indirectly by our application. We generally used a following command to generate profiles of our application:

```
nsys profile --sample=none \
             --trace=cuda,nvtx \
             --stats=true \
             <optional mpirun/srun> \
             <optional mps-wrapper> \
             app.py <app args>
```

The *nsys profile* launches and profiles our application. Usuaully, we disable CPU sampling (*--sample=none*) and only trace CUDA and NVTX APIs (*--trace=cuda,nvtx*) to limit noise in the profile output. When using MPI, we add the *mpirun* or equivalent (*srun* on NERSC systems) executable with its arguments following the arguments to the *nsys profile* segment of the command. Similarily, when using the CUDA Multi-Process Service, we include a wrapper shell script that ensures the service is launches and shutdowns from a single process per node. Finally, we specify the executable we wish to profile along with its arguments. The *--stats=true* option generates a set of useful summary statistics that is printed to stdout. For a more detailed look at runtime performance, it is useful view the generated report file using the NSight Systems GUI.

Nsight Systems provides a zoomable timeline view that allows us to visualize the performance of our code. Using Nsight Systems, we can see the regions of our code that we marked with NVTX wrappers, as well as the lower level memory and kernel operations. In Figure 3, we show a screenshot from an early profile of our GPU port using the NSight Systems GUI. At a high-level, we see that memory transfers and kernel executions, respectively, account for 3% and 97% of the time spent on GPU. From this profile, we identified that approximately 85% of the runtime of the application is spent in the "decorrelate" step of the algorithm. We also discovered an unexpected performance issue near the end patch extraction that we were able to solve using NumPy advanced array indexing. The execution time of the *decorrelate* method is dominated by the eigenvalue decomposition operations. Profiling also helped identify unexpected performance issues in code regions we did not expect.

Maximizing Node Utilization

We use multiple GPUs in our application via MPI (*mpi4py*). Since the CPU implementation is already using MPI, minimal refactor was required. Each MPI rank is assigned to a single GPU. Mapping MPI ranks to GPUs can be handled using slurm options (*--gpu-bind*), setting environment variables such as *CUDA_VISIBLE_DEVICES*, or at runtime using the CuPy API (*cupy.cuda.Device.use()*). We oversubscribe ranks to GPUs to saturate GPU utilization using CUDA Multi-Process Service (MPS), which allows kernel and memcopy operations from different processes to overlap on the GPU. Some care must be taken to avoid over allocating memory on each device. We use a shell script wrapper to ensure the CUDA MPS control daemon is started by a single process on each node process server before launching our application. At NERSC, we use the following script which references environment variables set by the slurm workload manager.

```
#!/bin/bash
# Example mps-wrapper usage:
# > srun -n 2 -c 1 mps-wrapper command arg1 ...
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log
# Launch MPS from a single rank per node
if [ $SLURM_LOCALID -eq 0 ]; then
    nvidia-cuda-mps-control -d
fi
# Wait for MPS to start
sleep 5
# Run the command
"$@"
# Quit MPS control daemon before exiting
if [ $SLURM_LOCALID -eq 0 ]; then
    echo quit | nvidia-cuda-mps-control
fi
```

In Figure 4, we show how performance scales with the number of GPUs used and the number of MPI ranks per GPU. The solid colored lines indicate the improved performance as we increase the number of GPU used. Different colors represent varying degrees of the number of MPI ranks per GPU. In this case, using 2 MPI ranks per GPU seems to saturate performance and we observe a slight degradation in performance oversubscribing further. We reached the GPU memory limit when attempting to go beyond 4 MPI ranks per GPU. The measurements for the analysis shown here were performed on test node at NERSC using 4 NVIDIA V100 GPUs. The Perlmutter system will use NVIDIA A100 (40GB) GPUs which have more cores and significantly more memory than the V100 (16GB) GPUs. A similar analysis showed that we could

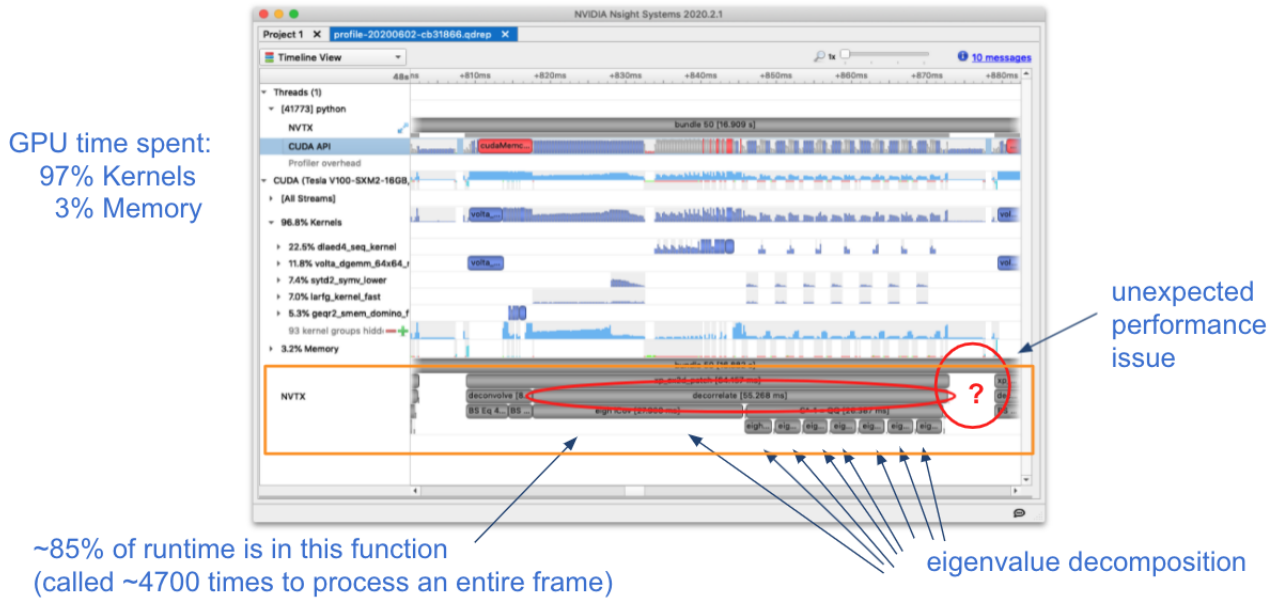

Fig. 3: *A screenshot of a profile from an early GPU port using NVIDIA Nsight Systems.*

go up to 5 MPI ranks per GPU on a test system with A100s. We note that while this configuration maximizes the expected GPU utilization on a Perlmutter with 4 A100 GPUs, the 64-core AMD Milan CPU is only at 31.25% utilization with 20 MPI ranks. Later on, we will discuss one way to utilize a few of these spare CPU cores.

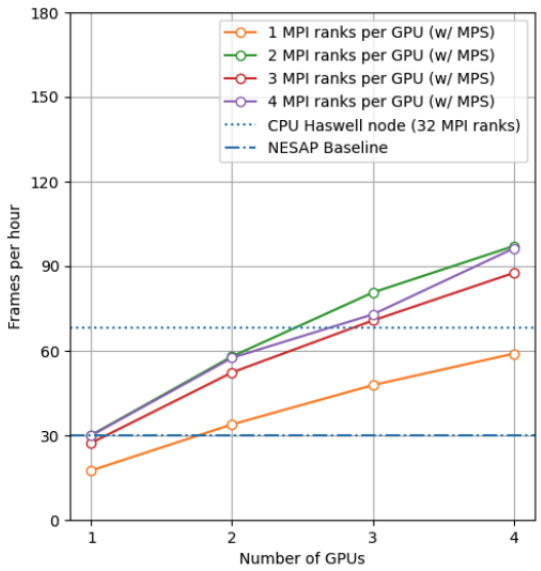

Fig. 4: *Performance scaling with multiple NVIDIA V100 GPUs. The solid colored lines indicate the improved performance as we increase the number of GPU used. Different colors represent varying degrees of the number of MPI ranks per GPU as indicated in the legend. The horizontal blue lines representing CPU-only measurements were approximate and only used for reference.*

Batching GPU Operations

Earlier, we observed that eigenvalue decomposition accounted for a significant portion of the execution time of our program. In the *spectro-perfectionism* algorithm, an eigenvalue decomposition is performed on the inverse covariance matrix which is then used to calculate the covariance matrix followed by several smaller eigenvalue decompositions that are performaned on the diagonal blocks of the covariance matrix. Since the small eigenvalue decompositions are performed on independent sub-matrices, we tried "batching" (or "stacking") the operations. We noted the existance of a *syevjBatched* function in CUDA cuSOLVER library which could perform eigenvalue decomposition on batches of input matrices using a Jacobi eigenvalue solver. This was not immediately available in Python via CuPy but we were able to implement Cython wrappers in CuPy using similar wrappers already present in CuPy as a guide. We submitted our implementation as a pull-request to the CuPy project on GitHub[5].

In Figure 5, we show profile snippets of that demonstate the improved performance using the Jacobi eigenvalue solvers from the cuSOLVER library. The execution time of the "decorrelate" method improved by a factor of two.

This inspired us to look for opportunities to use batched operations in our program. We found a significant speedup by refactoring the application to extract spectra from multiple patches in a subbundle using batched array and linear algebra operations. This allowed us to leverage batched Cholesky decomposition and solver operations on the GPU (*potrfBatched* and *potrsBatched* in the cuSOLVER library). We contributed *cupyx.linalg.posv* (named after LAPACK's xPOSV routines) to solve the linear equations $A\ x = b$ via Cholesky factorization of A, where A is a real symmetric or complex Hermitian positive-definite matrix[6]. Our implementation was essentially a generalization of an existing method *cupyx.linalg.invh*, which was implemented as the special case where the right-hand side of the equation is the Identity matrix. In Figure 6, we compare the profile timelines before and after implementing batch Cholesky decomposition and solver operations. The runtime for extraction over an entire subbundle of 5 spectra is 3.3 times faster using batched Cholesky operations.

5. https://github.com/cupy/cupy/pull/3488
6. https://github.com/cupy/cupy/pull/4291

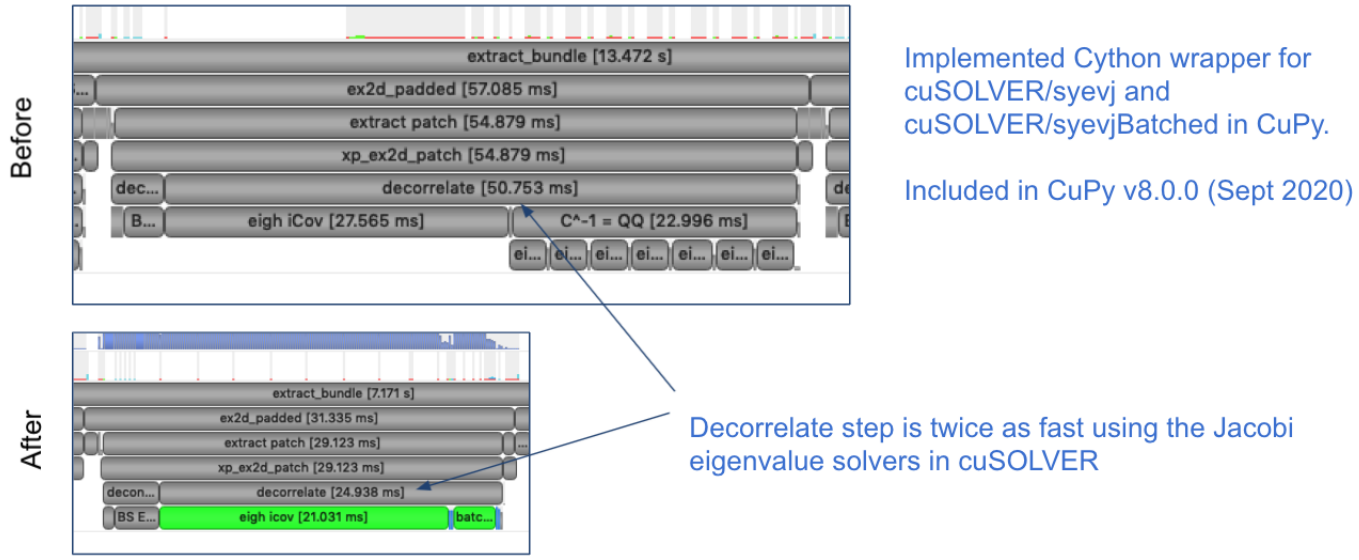

Fig. 5: *The "decorrelate" is twice as fast using the Jacobi eigenvalue solvers from the cuSOLVER library.*

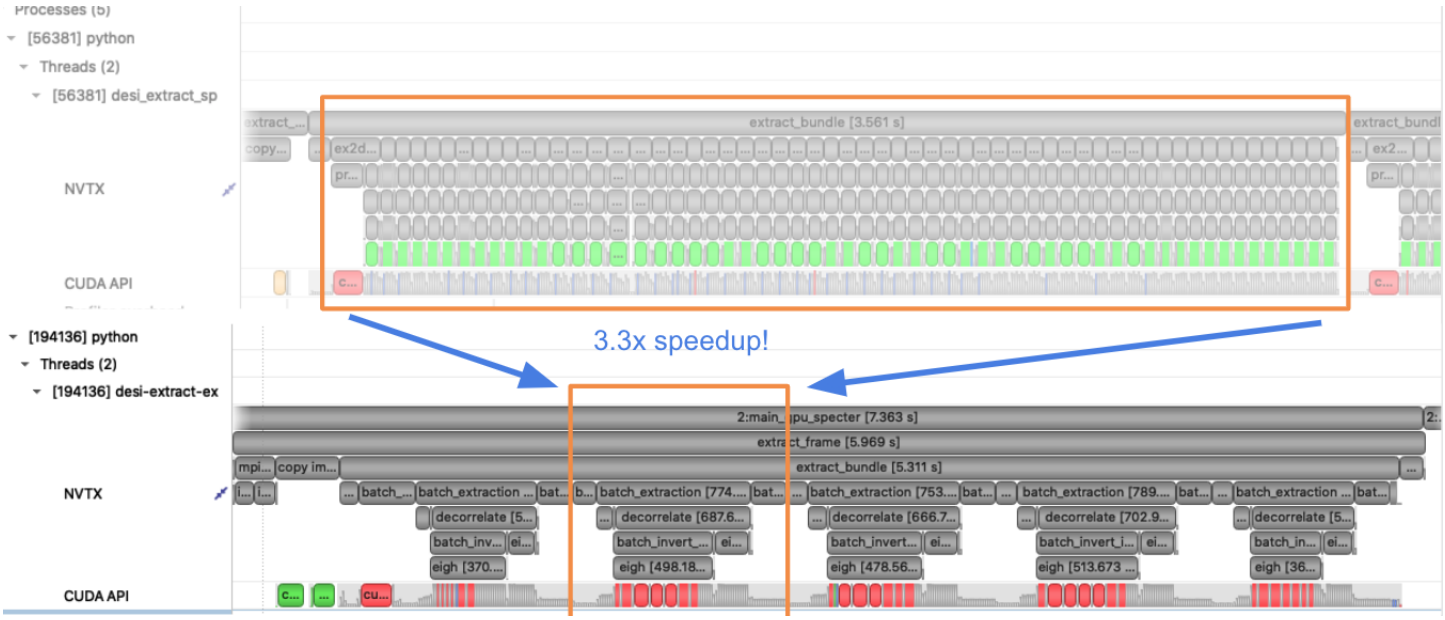

Fig. 6: *Profile demonstrating speedup from batch Cholesky solve.*

Overlapping Compute and IO

At this point, we observed that reading the input data and writing the output results accounted for approximately 25%-30% of the total wall time to process 30 frames from a single exposure in series using a single node. The input data is read by a single MPI rank, transferred to GPU memory, and then broadcast to other MPI ranks using CUDA-aware MPI. After extraction, each MPI rank transfers its results back to CPU memory and the results are gathered to the root MPI rank. The root MPI rank combines the results and writes the output to a FITS file on disk. Using spare CPU cores, we were able to hide most of this IO latency and better utilize the resources available on a node. When there are multiple frames processed per node, the write and read steps between successive frames can be interleaved with computation.

In Figure 7, we demonstrate how a subset of the MPI ranks communicate to achieve this functionality. At a high level, the processing of a single frame can be broken down into 3 distinct phases: read, work, and write. The frames are processed in series, frame one (green) is processed, then frame two (orange), and finally frame (three). Panel a shows the non-overlapping sequence of steps to process 3 frames in series. Panel b shows how the overlapping of IO and computation is orchestrated using two additional MPI ranks, dedicated reader and writer ranks. At the start of the program, the reader rank reads the input data while all worker ranks wait. The reader rank performs some initial preprocessing and sends the data to the root computation rank. Once the data has been sent, the reader rank begins reading the next frame. After the worker root receives the input data, it performs the work which can involve broadcasting the data to additional worker ranks in the computation group (not shown in

the diagram). The result on the root computation rank is then sent to a specially designated writer rank. The computation group ranks move on to processing the next frame which has already been read from disk by a specially designated read rank. Meanwhile, the writer rank finishes writing the previous result and is now waiting to receive the next result.

Overlapping compute and IO in this manner effectively hides the intermediate read and write operations between frames processed serially on a node, reducing the wall time by over 60 seconds and providing a 1.34x speedup in per-node throughput.

Results

Throughout development, we performed a standard benchmark after major feature implementations to track progress over time. For DESI, a useful and practical benchmark of performance is the number of frames that can be processed per node-time on NERSC systems. Specifically, we use the throughput measure *frames-per-node-hour* (FPNH) as the figure of merit (FoM) for this application. This figure enables DESI to cost out how much data it can process given a fixed allocation of compute resources.

A summary of benchmark results by major feature milestone is shown in Figure 8 and listed in Table 1. The benchmark uses data from a single exposure containing 30 CCD frames. After major feature implementations, we typically perform a scan of hyperparameter values to identify the optimal settings. For example, after the "batch-subbundle" implementation, the optimal number of wavelength bins per patch changed from 50 to 30. The baseline FoM for this application on the Edison and Cori supercomputers is 27.89 FPNH and 40.15 FPNH, respectively. The initial refactor improved the CPU-only performance on Cori Haswell by more than 50%. Our initial GPU port achieved 6.15 FPNH on Cori GPU nodes, an unimpressive mark compared to the baseline CPU benchmarks. Using visual profiling to guide optimization effort, we were able to iterively improve the performance to 362.2 FPNH on Cori GPU nodes.

Since the Perlmutter system is not available at the time of writing, we estimate the expected performance by running the benchmark on an NVIDIA DGX-A100 system. A Perlmutter GPU node will have the same NVIDIA A100 GPUs as the DGX system and the newer AMD Milan CPU compared to the AMD Rome CPU on DGX. The projected FoM for this application on the new Perlmutter supercomputer is 575.25 FPNH, a roughly 20x improvement over the Edison baseline.

Going forward, the team will need to re-evaluate where to refocus optimization efforts. The performance of the spectral extraction step is now comparable to other steps in the DESI data processing pipeline. We are currently evaluating other steps in the DESI pipeline for GPU acceleration. The DESI team may also opt to spend the improved efficiency to perform more compute intensive processing if there is a scientific opportunity.

Conclusion

The rising popularity of heterogenous CPU-GPU computing platforms offers an opportunity for improving the performance of science applications. Adapting scientific Python applications to use GPU devices is relatively seamless due to the community of developers working on GPU-accelerated libraries that provide Numpy-compatible and SciPy-compataible APIs and, of course, the excellent foundation provided by NumPy and SciPy projects. Profiling tools such as NVIDA Nsight Systems and the *cProfile* Python module often provide actionable insights to that can focus optimization efforts. Refactoring code to expose parallelism and use more vectorized operations often improves performance on both CPU and GPU computing architectures. For DESI, the transition to GPUs on Perlmutter will shorten the time it takes to process years worth of data from weeks to months down to hours to days.

*. Note that the *initial-gpu* benchmark only processed a single frame instead of all 30 frames from an expoosure.

Acknowledgements

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

This research is supported by the Director, Office of Science, Office of High Energy Physics of the U.S. Department of Energy under Contract No. DE–AC02–05CH11231, and by the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility under the same contract; additional support for DESI is provided by the U.S. National Science Foundation, Division of Astronomical Sciences under Contract No. AST-0950945 to the NSF's National Optical-Infrared Astronomy Research Laboratory; the Science and Technologies Facilities Council of the United Kingdom; the Gordon and Betty Moore Foundation; the Heising-Simons Foundation; the French Alternative Energies and Atomic Energy Commission (CEA); the National Council of Science and Technology of Mexico; the Ministry of Economy of Spain, and by the DESI Member Institutions. The authors are honored to be permitted to conduct astronomical research on Iolkam Du'ag (Kitt Peak), a mountain with particular significance to the Tohono O'odham Nation.

References

[BS10] Adam S. Bolton and David J. Schlegel. Spectro-Perfectionism: An Algorithmic Framework for Photon Noise-Limited Extraction of Optical Fiber Spectroscopy. *Publications of the Astronomical Society of the Pacific*, 122(888):248, February 2010. doi:10.1086/651008.

[DPS05] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005. doi:10.1016/j.jpdc.2005.03.010.

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

[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.

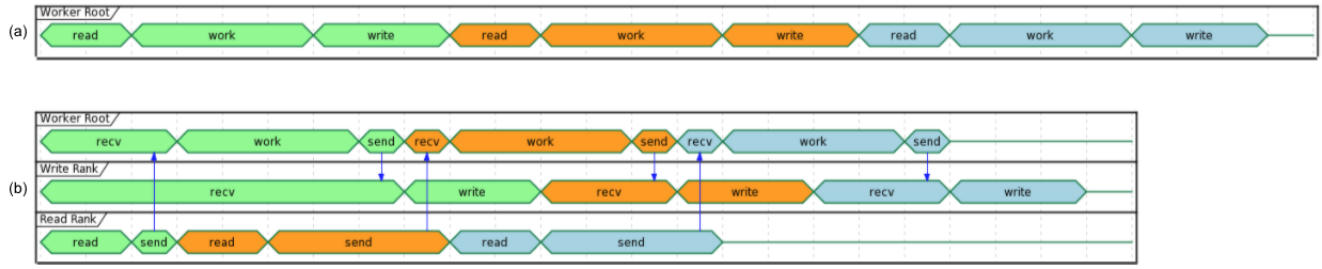

Fig. 7: *Overlapping IO and compute. In panel a, we show an example timeline of the root worker MPI rank performing the read, work, and write steps to process 3 frames. In panel b, we show an example timeline of the root worker, read, and write MPI ranks performing the read, work, and write steps along with their inter-communication to process 3 frames.*

Note	System	Arch (CPU/GPU)	Nodes	GPUs Per Node	MPI Ranks Per Node	Walltime (sec)	FPNH
baseline	Edison	Xeon	25	-	24	154.9	27.89
	Cori	Haswell	19	-	32	141.6	40.15
cpu-refactor	Cori	Haswell	2	-	32	830.2	65.05
initial-gpu	CoriGPU	Skylake/V100	1	1	1	585.5*	6.15
multi-gpu	CoriGPU	Skylake/V100	2	4	8	611.6	88.30
	DGX	Rome/A100	2	4	16	526.8	102.51
batch-eigh	CoriGPU	Skylake/V100	2	4	8	463.7	116.46
	DGX	Rome/A100	2	4	16	372.7	144.90
batch-subbundle	CoriGPU	Skylake/V100	1	4	8	458.9	235.36
	DGX	Rome/A100	1	4	20	252.4	427.86
interleave-io	CoriGPU	Skylake/V100	1	4	10	362.2	298.19
	DGX	Rome/A100	1	4	22	187.7	575.25

TABLE 1: *Summary of benchmark results by major feature milestone.*

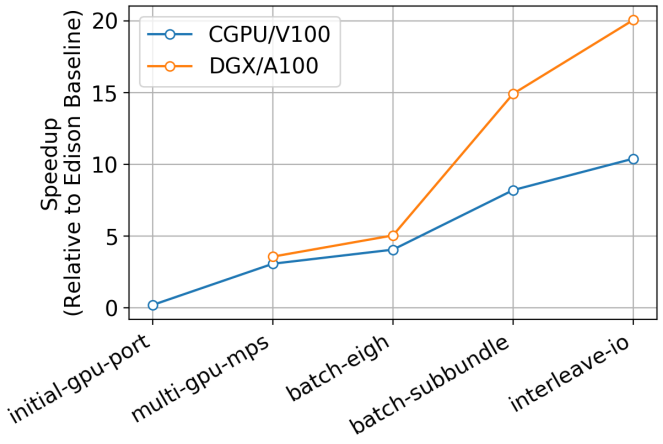

Fig. 8: *DESI Figure-of-Merit progress by major feature milestone.*

[OUN+17] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[RTD+17] Zahra Ronaghi, Rollin Thomas, Jack Deslippe, Stephen Bailey, Doga Gursoy, Theodore Kisner, Reijo Keskitalo, and Julian Borrill. Python in the NERSC Exascale Science Applications Program for Data. In *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*, PyHPC'17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3149869.3149873.

[STB19] Laurie A. Stephey, Rollin C. Thomas, and Stephen J. Bailey. Optimizing Python-Based Spectroscopic Data Processing on NERSC Supercomputers. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 69 – 76, 2019. doi:10.25080/Majora-7ddc1dd1-00a.

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

MPI-parallel Molecular Dynamics Trajectory Analysis with the H5MD Format in the MDAnalysis Python Package

Edis Jakupovic[‡], Oliver Beckstein[‡*]

Abstract—Molecular dynamics (MD) computer simulations help elucidate details of the molecular processes in complex biological systems, from protein dynamics to drug discovery. One major issue is that these MD simulation files are now commonly terabytes in size, which means analyzing the data from these files becomes a painstakingly expensive task. In the age of national supercomputers, methods of parallel analysis are becoming a necessity for the efficient use of time and high performance computing (HPC) resources but for any approach to parallel analysis, simply reading the file from disk becomes the performance bottleneck that limits overall analysis speed. One promising way around this file I/O hurdle is to use a parallel message passing interface (MPI) implementation with the HDF5 (Hierarchical Data Format 5) file format to access a single file simultaneously with numerous processes on a parallel file system. Our previous feasibility study suggested that this combination can lead to favorable parallel scaling with hundreds of CPU cores, so we implemented a fast and user-friendly HDF5 reader (the `H5MDReader` class) that adheres to H5MD (HDF5 for Molecular Dynamics) specifications. We made `H5MDReader` (together with a H5MD output class `H5MDWriter`) available in the MDAnalysis library, a Python package that simplifies the process of reading and writing various popular MD file formats by providing a streamlined user-interface that is independent of any specific file format. We benchmarked H5MDReader's parallel file reading capabilities on three HPC clusters: ASU Agave, SDSC Comet, and PSC Bridges. The benchmark consisted of a simple split-apply-combine scheme of an I/O bound task that split a 90k frame (113 GiB) coordinate trajectory into N chunks for N processes, where each process performed the commonly used RMSD (root mean square distance after optimal structural superposition) calculation on their chunk of data, and then gathered the results back to the root process. For baseline performance, we found maximum I/O speedups at 2 full nodes, with Agave showing 20x, and a maximum computation speedup on Comet of 373x on 384 cores (all three HPCs scaled well in their computation task). We went on to test a series of optimizations attempting to speed up I/O performance, including adjusting file system stripe count, implementing a masked array feature that only loads relevant data for the computation task, front loading all I/O by loading the entire trajectory into memory, and manually adjusting the HDF5 dataset chunk shapes. We found the largest improvement in I/O performance by optimizing the chunk shape of the HDF5 datasets to match the iterative access pattern of our analysis benchmark. With respect to baseline serial performance, our best result was a 98x speedup at 112 cores on ASU Agave. In terms of absolute time saved, the analysis went from 4623 seconds in the baseline serial run to 47 seconds in the parallel, properly chunked run. Our results emphasize the fact that file I/O is not just dependent on the access pattern of the file, but more so the synergy between access pattern and the layout of the file on disk.

Index Terms—Molecular Dynamics Simulations, High Performance Computing, Python, MDAnalysis, HDF5, H5MD, MPI I/O

‡ *Arizona State University*
* *Corresponding author: obeckste@asu.edu*

Introduction

The molecular dynamics (MD) simulation approach [HBD+19] is widely used across the biomolecular and materials sciences, accounting for more than one quarter of the total computing time [FQC+19] in the Extreme Science and Engineering Discovery Environment (XSEDE) network of national supercomputers in the US [TCD+14]. MD simulations, especially in the realm of studying protein dynamics, serve an important purpose in characterizing the dynamics, and ultimately the function of a protein [Oro14]. For example, recent award-winning work [CDG+21] involving the SARS-CoV-2 spike protein was able to use all-atom MD simulations to elucidate the dynamics of the virus-to-human cell interaction that was inaccessible to experiment. While the parameters involved in fine tuning the physics driving these simulations continue to improve, the computational demand of longer, more accurate simulations increases [DDG+12]. As high performance computing (HPC) resources continue to improve in performance, the size of MD simulation files are now commonly terabytes in size, making serial analysis of these trajectory files impractical [CR15]. Parallel analysis is a necessity for the efficient use of both HPC resources and a scientist's time [BFJ18], [FQC+19]. MD trajectory analysis can be parallelized using task-based or MPI-based (message passing interface) approaches, each with their own advantages and disadvantages [PLK+18]. Here we investigate parallel trajectory analysis with the MDAnalysis Python library [MADWB11], [GLB+16]. MDAnalysis is a widely used package in the molecular simulation community that can read and write over 25 popular MD trajectory file formats while providing a common object-oriented interface that makes data available as numpy arrays [HMvdW+20]. MDAnalysis aims to bridge the entrenched user communities of different MD packages, allowing scientists to more easily (and productively) move between these entrenched communities. Previous work that focused on developing a task-based approach to parallel analysis found that an I/O bound task only scaled to 12 cores due to a file I/O bottleneck [SFMLIP+19]. Our recent feasibility study suggested that parallel reading via MPI-IO and the HDF5 file format could

lead to good scaling although only a reduced size custom HDF5 trajectory was investigated and no usable implementation of a true MD trajectory reader was provided [KPF+20].

H5MD, or "HDF5 for molecular data", is an HDF5-based file format that is used to store MD simulation data, such as particle coordinates, box dimensions, and thermodynamic observables [dBCH14]. A Python reference implementation for H5MD exists (pyh5md [dBCH14]) but the library is not maintained anymore, and with advice from the original author of pyh5md, we implemented native support for H5MD I/O in the MDAnalysis package. HDF5 is a structured, binary file format that organizes data into two objects: groups and datasets. It implements a hierarchical, tree-like structure, where groups represent nodes of the tree, and datasets represent the leaves [Col14]. An HDF5 file's datasets can be stored either contiguously on disk, or scattered across the disk in different locations in *chunks*. These chunks must be defined on initialization of the dataset, and for any element to be read from a chunk, the entire chunk must be read. The HDF5 library can be built on top of a message passing interface (MPI) implementation so that a file can be accessed in parallel on a parallel file system such as Lustre or BeeGFS. We implemented a parallel MPI-IO capable HDF5-based file format trajectory reader into MDAnalysis, H5MDReader, that adheres to the H5MD specifications. H5MDReader interfaces with h5py, a high level Python package that provides a Pythonic interface to the HDF5 format [Col14]. In `h5py`, accessing a file in parallel is accomplished by passing a keyword argument into `h5py.File`, which then manages parallel disk access.

The BeeGFS and Lustre parallel file systems are well suited for multi-node MPI parallelization. One key feature of a Lustre parallel file systems is file striping, which is the ability to store data from a file across multiple physical locations, known as object storage targets (OSTs), where "stripe count" refers to the number of OSTs to which a single file is striped across. Thinking carefully about the synchronization of chunk shape and stripe settings can be crucial to establishing optimal I/O performance [How10]. We tested various algorithmic optimizations for our benchmark, including using various stripe counts (1, 48, 96), loading only necessary coordinate information with numpy masked arrays [HMvdW+20], and front loading all I/O by loading the entire trajectory chunk into memory prior to the RMSD calculation.

We benchmarked H5MDReader's parallel reading capabilities with MDAnalysis on three HPC clusters: ASU Agave at Arizona State University, and SDSC Comet and PSC Bridges, which are part of XSEDE [TCD+14]. The benchmark consisted of a simple split-apply-combine scheme [Wic11] of an I/O-bound task that split a 90k frame (113 GiB) trajectory into N chunks for N processes, where each process performed a computation on their chunk of data, and the results were finally gathered back to the root process. For the computational task, we computed the time series of the root mean squared distance (RMSD) of the positions of the C_α (alpha carbon) atoms in the protein to their initial coordinates at the first frame of the trajectory. At each frame (time step) in the trajectory, the protein was optimally superimposed on the reference frame to remove translations and rotations. The RMSD calculation is a very common task performed to analyze the dynamics of the structure of a protein [MM14]. Because it is a fast computation that is bounded by how quickly data can be read from the file it is a suitable task to test the I/O capabilities of H5MDReader.

We tested the effects of HDF5 file chunking and file compression on I/O performance. In general we found that altering the stripe count and loading only necessary coordinates via masked arrays provided little improvement in benchmark times. Loading the entire trajectory into memory in one pass instead of iterating through, frame by frame, showed the greatest improvement in performance. This was compounded by our results with HDF5 chunking. Our baseline test file was auto-chunked with the auto-chunking algorithm in `h5py`. When we recast the file into a contiguous form and a custom, optimized chunk layout, we saw improvements in serial I/O on the order of 10x. Additionally, our results from applying gzip compression to the file showed no loss in performance at higher processor counts, indicating H5MD files can be compressed without losing performance in parallel analysis tasks.

Methods

HPC environments

We tested the parallel MPI I/O capabilities of our H5MD implementation on three supercomputing environments: ASU Agave, PSC Bridges, and SDSC Comet. The Agave supercomputer offers 498 compute nodes. We utilized the Parallel Compute Nodes that offer 2 Intel Xeon E5-2680 v4 CPUs (2.40GHz, 14 cores/CPU, 28 cores/node, 128GB RAM/node) with a 1.2PB scratch BeeGFS file system that uses an Intel OmniPath interconnect system. The Bridges supercomputer offers over 850 compute nodes that supply 1.3018 Pf/s and 274 TiB RAM. We utilized the Regular Shared Memory Nodes that offer 2 Intel Haswell E5-2695 v3 CPUs (2.3-3.3GHz, 14 cores/CPU, 28 cores/node, 128GB RAM/node) with a 10PB scratch Lustre parallel file system that uses an InfiniBand interconnect system. The Comet supercomputer offers 2 Pf/s with 1944 standard compute nodes. We utilized the Intel Haswell Standard Compute Nodes that offer 2 Intel Xeon E5-2680 v3 CPUs (2.5GHz, 12 cores/CPU, 24 cores/node, 128GB RAM/node) with a 13PB scratch Lustre parallel file system that also uses an InfiniBand interconnect system.

Our software library stacks were built with conda environments. Table 1 gives the versions of each library involved in the stack. We used GNU C compilers on Agave and Bridges and the Intel C-compiler on Comet for MPI parallel jobs as recommended by the Comet user guide. We used OpenMPI as the MPI implementation on all HPC resources as this was generally the recommended environment and in the past we found it also the easiest to build against [KPF+20]. The mpi4py [DPKC11] package was used to make MPI available in Python code, as required by `h5py`. In general, our software stacks were built in the following manner:

- module load anaconda3
- create new conda environment
- module load parallel hdf5 build
- module load OpenMPI implementation
- install mpi4py with `env MPICC=/path/to/mpicc pip install mpi4py`
- install h5py with `CC="mpicc" HDF5_MPI="ON" HDF5_DIR=/path/to/parallel-hdf5 pip install --no-binary=h5py h5py`
- install development MDAnalysis as outlined in the MDAnalysis User Guide

System	ASU Agave	PSC Bridges	SDSC Comet
Python	3.8.5	3.8.5	3.6.9
C compiler	gcc 4.8.5	gcc 4.8.5	icc 18.0.1
HDF5	1.10.1	1.10.2	1.10.3
OpenMPI	3.0.0	3.0.0	3.1.4
h5py	2.9.0	3.1.0	3.1.0
mpi4py	3.0.3	3.0.3	3.0.3
MDAnalysis	2.0.0-dev0	2.0.0-dev0	2.0.0-dev0

TABLE 1: *Library versions installed for each HPC environment.*

name	format	file size (GiB)
H5MD-default	H5MD	113
H5MD-chunked	H5MD	113
H5MD-contiguous	H5MD	113
H5MD-gzipx1	H5MD	77
H5MD-gzipx9	H5MD	75
DCD	DCD	113
XTC	XTC	35
TRR	TRR	113

TABLE 2: *Data files benchmarked on all three HPCS.* ***name*** *is the name that is used to identify the file in this paper.* ***format*** *is the format of the file, and* ***file size*** *gives the size of the file in gibibytes.* ***H5MD-default*** *original data file written with pyh5md which uses the auto-chunking algorithm in h5py.* ***H5MD-chunked*** *is the same file but written with chunk size (1, n atoms, 3) and* ***H5MD-contiguous*** *is the same file but written with no HDF5 chunking.* ***H5MD-gzipx1*** *and* ***H5MD-gzipx9*** *have the same chunk arrangement as* ***H5MD-chunked*** *but are written with gzip compression where 1 is the lowest level of compression and 9 is the highest level.* ***DCD****,* ***XTC****, and* ***TRR*** *are copies* ***H5MD-contiguous*** *written with MDAnalysis.*

Benchmark Data Files

The test data files used in our benchmark consist of a topology file `YiiP_system.pdb` with 111,815 atoms and a trajectory file `YiiP_system_9ns_center100x.h5md` with 90100 frames. The initial trajectory data file (H5MD-default in Table 2) was generated with pyh5md [dBCH14] using the XTC file `YiiP_system_9ns_center.xtc` [SFMLIP+19], [LRFK+21], using the "ChainReader" facility in MDAnalysis with the list `100 * ["YiiP_system_9ns_center.xtc"]` as input. The rest of the test files were copies of H5MD-default and were written with MDAnalysis using different HDF5 chunking arrangements and compression settings. Table 2 gives all of the files benchmarked with how they are identified in this paper as well as their corresponding file size.

Parallel Algorithm Benchmark

We implemented a simple split-apply-combine parallelization algorithm [Wic11], [SFMLIP+19], [KPF+20] that divides the number of frames in the trajectory evenly among all available processes. Each process receives a unique `start` and `stop` for which to iterate through their section of the trajectory. As the computational task, the root mean square distance (RMSD) of the protein C_α atoms after optimal structural superposition [MM14] is computed at each frame with the QCProt algorithm [The05], as described in our previous work [SFMLIP+19], [KPF+20].

In order to obtain detailed timing information we instrumented code as follows below. Table 3 outlines the specific lines in the code that were timed in the benchmark.

```
1 import MDAnalysis as mda
2 from MDAnalysis.analysis.rms import rmsd
3 from mpi4py import MPI
```

line number	id	description
11	$t^{\text{init\_top}}$	load topology file
12	$t^{\text{init\_traj}}$	load trajectory file
38	$t^{\text{I/O}}$	read data from time step into memory
39	t^{compute}	perform rmsd computation
42	t^{wait}	wait for processes to synchronize
47	$t^{\text{comm\_gather}}$	combine results back into root process

TABLE 3: *All timings collected from the example benchmark code.* ***id*** *gives the reference name used in this paper to reference the corresponding line number and timing collected.* ***description*** *gives a short description of what that specific line of code is doing in the benchmark.*

```
4 import numpy as np
5
6 comm = MPI.COMM_WORLD
7 size = comm.Get_size()
8 rank = comm.Get_rank()
9
10 def benchmark(topology, trajectory):
11     u = mda.Universe(topology)
12     u.load_new(trajectory,
13                driver="mpio",
14                comm=comm)
15     CA = u.select_atoms("protein and name CA")
16     x_ref = CA.positions.copy()
17
18     # make_balanced_slices divides n_frames into
19     # equally sized blocks and returns start:stop
20     # indices for each block
21     slices = make_balanced_slices(n_frames,
22                                   size,
23                                   start=0,
24                                   stop=n_frames,
25                                   step=1)
26      start = slices[rank].start
27      stop = slices[rank].stop
28      bsize = stop - start
29
30      # sendcounts is used for Gatherv() to know how
31      # many elements are sent from each rank
32      sendcounts = np.array([
33          slices[i].stop - slices[i].start
34          for i in range(size)])
35
36      rmsd_array = np.empty(bsize, dtype=float)
37      for i, frame in enumerate(range(start, stop)):
38         ts = u.trajectory[frame]
39         rmsd_array[i] = rmsd(CA.positions,
40                              x_ref,
41                              superposition=True)
42      comm.Barrier()
43      rmsd_buffer = None
44      if rank == 0:
45         rmsd_buffer = np.empty(n_frames,
46                                dtype=float)
47      comm.Gatherv(sendbuf=rmsd_array,
48                   recvbuf=(rmsd_buffer, sendcounts), root=0)
```

The HDF5 file is opened with the `mpio` driver and the `MPI.COMM_WORLD` communicator to ensure the file is accessed in parallel via MPI I/O. The topology and trajectory initialization times must be analyzed separately because the topology file is not opened in parallel and represents a fixed cost each process must pay to open the file. MDAnalysis reads data from MD trajectory files one frame, or "snapshot" at a time. Each time the `u.trajectory[frame]` is iterated through, MDAnalysis reads the file and fills in numpy arrays [HMvdW+20] corresponding to that time step. Each MPI process runs an identical copy of the script, but receives a unique `start` and `stop` variable

such that the entire file is read in parallel. Gathering the results is done collectively by MPI, which means all processes must finish their iteration blocks before the results can be returned. Therefore, it is important to measure t^{wait} as it represents the existence of "straggling" processes. If one process takes substantially longer than the others to finish its iteration block, all processes are slowed down. These 6 timings are returned and saved as an array for each benchmark run.

We applied this benchmark scheme to H5MD test files on Agave, Bridges, and Comet. Each benchmark run received a unique, freshly copied test file that was only used once so as to avoid any caching effects of the operating system or file system. We also tested three algorithmic optimizations: Lustre file striping, loading the entire trajectory into memory, and using masked arrays in numpy to only load the C_α coordinates required for the RMSD calculation. For striping, we ran the benchmark on Bridges and Comet with a file stripe count of 48 and 96. For the into memory optimization, we used `MDAnalysis.Universe.transfer_to_memory()` to read the entire file in one go and pass all file I/O to the HDF5 library. For the masked array optimization, we allowed `u.load_new()` to take a list or array of atom indices as an argument, `sub`, so that the `MDAnalysis.Universe.trajectory.ts` arrays are instead initialized as `numpy.ma.masked_array` instances and only the indices corresponding to `sub` are read from the file.

Performance was quantified by measuring the I/O timing returned from the benchmarks, and strong scaling was assessed by calculating the speedup $S(N) = t_1/t_N$ and the efficiency $E(N) = S(N)/N$.

Data Sharing

All of our SLURM submission shell scripts and Python benchmark scripts for all three HPC environments are available in the repository https://github.com/Becksteinlab/scipy2021-mpiH5MD-data and are archived under DOI 10.5281/zenodo.5083858.

Results and Discussion

Baseline Benchmarks

We first ran benchmarks with the simplest parallelization scheme of splitting the frames of the trajectory evenly among all participating processes. The H5MD file involved in the benchmarks was written with the pyh5md library, the original Python reference implementation for the H5MD format [dBCH14]. The datasets in the data file were chunked automatically by the auto-chunking algorithm in `h5py`. File I/O remains the largest contributor to the total benchmark time, as shown by Figure 1 (A). Figure 1 (B, D-F) also show that the initialization, computation, and MPI communication times are negligible with regards to the overall analysis time. t^{wait}, however, becomes increasingly relevant as the number of processes increases (Figure 1 C), indicating a growing variance in the iteration block time across all processes. In effect, t^{wait} is measuring the occurrence of "straggling" processes, which has been previously observed to be an issue on busy, multi-user HPC environments [KPF+20]. We found that the total benchmark time continues to decrease as the number of processes increases to over 100 (from 4648 ± 319 seconds at $N = 1$ to 315.6 ± 59.8 seconds at $N = 112$ on Agave) (Fig. 2 A). While the absolute time of each benchmark is important in terms of measuring the actual amount of time saved with our parallelization scheme, results are

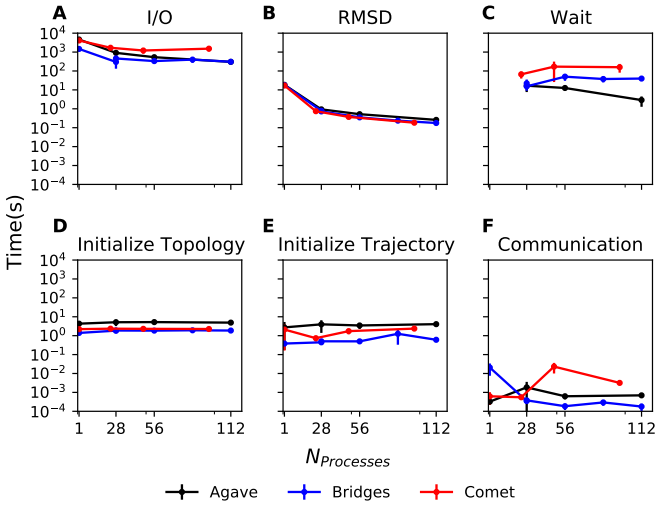

Fig. 1: *Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters. The benchmark was run on up to 4 full nodes on each HPC, where $N_{processes}$ was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The `H5MD-default` file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

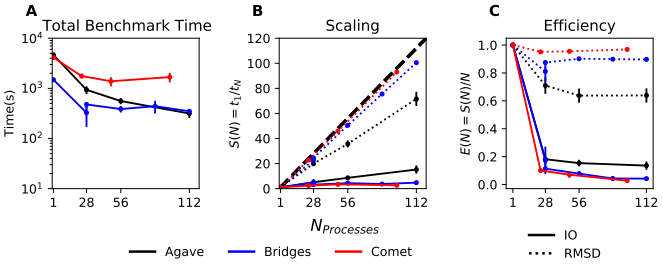

Fig. 2: *Strong scaling I/O and RMSD performance of the RMSD analysis task of the `H5MD-default` data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

often highly variable in a crowded HPC environment [How10] and therefore we focus our analysis on the speedup and efficiency of each benchmark run. The maximum total I/O speedup observed is only 15x and efficiencies at around 0.2 (Fig. 2 B, C). The RMSD computation scaling, on the other hand, remains high, with nearly ideal scaling on Bridges and Comet, with Agave trailing behind at 71x speedup at 122 cores. Therefore, for a computationally bound analysis task, our parallel H5MD implementation will likely scale well.

Effects of Algorithmic Optimizations on File I/O

We tested three optimizations aimed at shortening file I/O time for the same data file. In an attempt to optimize I/O, we tried to minimize "wasted I/O". For example, in any analysis task, not all coordinates in the trajectory may be necessary for the computation. In our analysis test case, the RMSD was calculated for only the C_α atoms of the protein backbone, therefore the

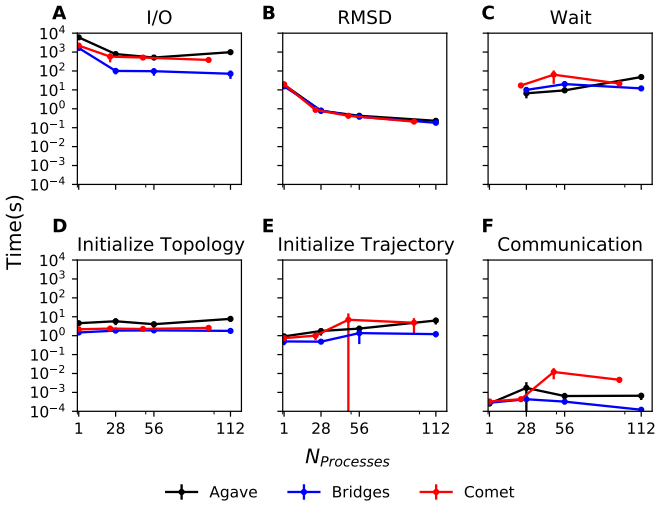

Fig. 3: Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters for the `masked_array` optimization technique. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The `H5MD-default` file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.

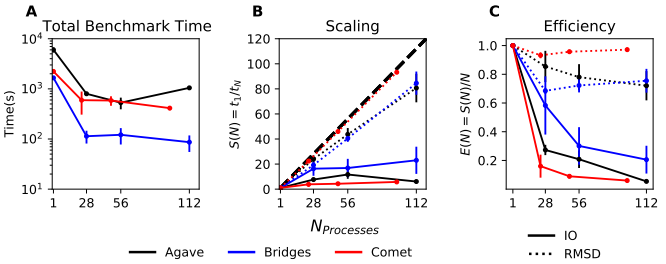

Fig. 4: Strong scaling performance of the RMSD analysis task with the `masked_array` optimization technique. The benchmark used the `H5MD-default` data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.

coordinates of all other atoms read from the file is essentially wasted I/O. To circumvent this issue, we implemented the use of NumPy `ma.masked_array` [HMvdW+20], where the arrays of coordinate data are instead initialized as masked arrays that only fill data from selected coordinate indices. We found that Bridges showed the best scaling with the masked array implementation, with a total scaling of 23x at 4 full nodes (1642 $\pm$ 115 seconds at $N = 1$ to 71 $\pm$ 33 seconds at $N = 112$ cores) as seen in Figure 4 (A, B). Agave showed a maximum scaling of 11x at 2 full nodes, while Comet showed 5x scaling at 4 full nodes (Figure 4 B). In some cases, the masked array implementation resulted in slower I/O times. For example, Agave went from 4623 seconds in the baseline 1 core run to 5991 seconds with masked arrays. This could be due to the HDF5 library not being optimized to work with masked arrays as with numpy arrays. On the other hand, for Bridges and Comet, we observed an approximate 5x speedup in I/O time (Fig. 4 B) for the masked array case when compared to the baseline benchmark. In terms of the RMSD computation scaling, we once again found all three systems scaled well, with Comet displaying ideal scaling all the way to 4 full nodes, while Agave and Bridges hovering around 85x at 112 cores.

With an MPI implementation, processes participating in parallel I/O communicate with one another. It is commonly understood that repeated, small file reads performs worse than a large, contiguous read of data. With this in mind, we tested this concept in our benchmark by loading the entire trajectory into memory prior to the RMSD task. Modern super computers make this possible as they contain hundreds of GiB of memory per node. On Bridges, loading into memory strangely resulted in slower I/O times (1466s baseline to 2196s at $N = 1$ and 307s baseline to 523s at $N = 112$, Fig. 1 A and Fig. 5 A). Agave and Comet, on the other hand, showed surprisingly different results. They both performed substantially better for the $N = 1$ core case. Agave's serial I/O performance was boosted from 4623s to 891s (Fig. 5 A) by loading the data into memory in one slurp rather than iterating through the trajectory frame by frame. Similarly, Comet's serial I/O performance went from 4101s to 1740s, with multi-node performance continuing to show improvement versus the baseline numbers (excluding the peak at $N = 48$). Agave steady improvements in performance all the way to 4 full nodes, where the I/O time reached 73s (Fig. 5 A, Fig. 6 A). Figure 7 gives a direct comparison on Agave of the baseline benchmark performance with the two optimization methods outlined. With respect to the baseline serial performance, loading into memory gives a 91x speedup (4658s at 1 core to 73s at 112 cores) (Figure 7, A). This result was interesting in that the only difference between the two was the access pattern of the data - in one case, the file was read in small repeated bursts, while in the other the file was read from start to finish with HDF5. We hypothesized that this was due to layout of the file itself on disk.

Also, we found that the t^{wait} does not increase as the number of processes increases as in all of the other benchmark cases (Figure 5 C). In the other benchmarks, t^{wait} was typically on the order of 10-200 seconds, whereas t^{wait} on the order of 0.01 seconds for the memory benchmarks (Figure 7 C). This indicates that the cause of the iteration block time variance among processes stems from MPI rank coordination when many small read requests are made.

To investigate MPI rank competition, we increased the stripe count on Bridge's and Comet's Lustre file system up to 96, where found marginal I/O scaling improvements of 1.2x on up to 4 full nodes (not shown). While our data showed no improvement with altering the stripe count, this may have been a byproduct the poor chunk layout of the original file on disk. In the next section we discuss the effects of HDF5 chunking on I/O performance.

Effects of HDF5 Chunking on File I/O

To test the hypothesis that the increase in serial file I/O between the baseline performance in loading into memory performance was caused by the layout of the file on disk, we created *H5MDWriter*, an MDAnalysis file format writer class that gives one the ability to write H5MD files with the MDAnalysis user interface. These files can be written with user-decided custom chunk layouts, file compression settings, and can be opened with MPI parallel drivers that enable parallel writing. We ran some initial serial writing tests and found that writing from DCD, TRR, and XTC to H5MD

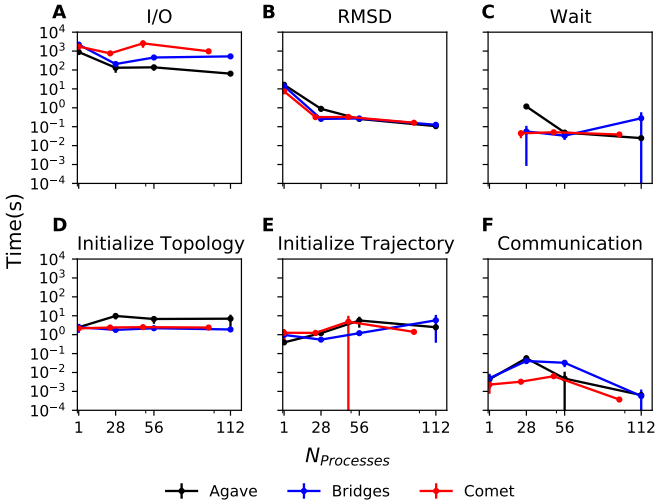

Fig. 5: *Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters for the loading-into-memory optimization technique. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The* `H5MD-default` *file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

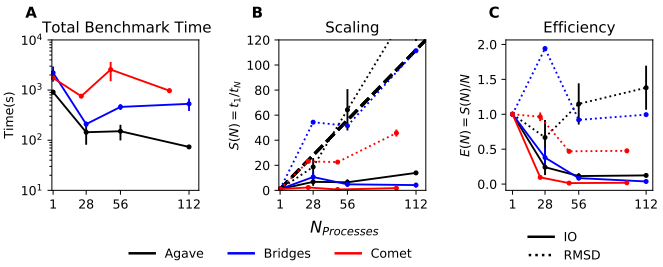

Fig. 6: *Strong scaling I/O performance of the RMSD analysis task with the loading-into-memory optimization technique. The benchmark used the* `H5MD-default` *data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

typically took ~360 seconds on Agave. For the 113 GiB test file, this was a 0.31 GiB/s write bandwidth. We rewrote the H5MD-default test file and tested two cases: one in which the file is written with no chunking applied (H5MD-contiguous), and one in which we applied a custom chunk layout to match the access pattern on the file (H5MD-chunked). Our benchmark follows a common MD trajectory analysis scheme in that it iterates through the trajectory one frame at a time. Therefore, we applied a chunk shape of `(1, n atoms, 3)` which matched exactly the shape of data to be read at each iteration step. An important feature of HDF5 chunking to note is that, for any element in a chunk to be read, the **entire** chunk must be read. When we investigated the chunk shape of the H5MD-default that was auto-chunked with h5py's chunking algorithm, we found that each chunk contained data elements from multiple different time steps. This means,

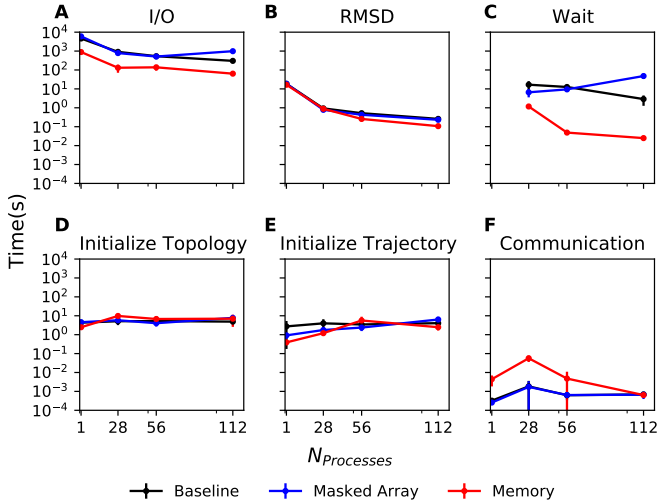

Fig. 7: *Benchmark timings on ASU Agave comparing the baseline benchmark with the masked array and loading into memory optimizations. Each benchmark was run on up to 4 full nodes where N processes was 1, 28, 56, and 112. The* `H5MD-default` *test file was used in all benchmarks. Points represent the mean over three repeats with the standard deviation shown as error bars.*

for every time step of data read, an exorbitant amount of excess data was being read and discarded at each iteration step. Before approaching the parallel tests, we tested how the chunk layout affects baseline serial I/O performance for the file. We found I/O performance strongly depends on the chunk layout of the file on disk. The auto-chunked H5MD-default file I/O time was 4101s, while our custom chunk layout resulted in an I/O time of 460s (Figure 8). So, we effectively saw a 10x speedup just from optimizing the chunk layout alone, where even the file with no chunking applied showed similar improvements in performance. In our previous serial I/O tests, we found that H5MD performed worse than other file formats, so we repeated those tests with our custom chunked file, H5MD-chunked. We found for our test file of 111,815 atoms and 90100 frames, H5MD outperformed XTC and TRR, while performing equally well to the DCD file, an encouraging result (Fig. 9).

Next, we investigated what effect the chunk layout had on parallel I/O performance. We repeated our benchmarks on Agave (at this point, Bridges had been decommissioned and our Comet allocation had expired) but with the H5MD-chunked and H5MD-contiguous data files. For the serial one process case, we found a similar result in that the I/O time was dramatically increased with an approximate 10x speedup for both the contiguous and chunked file, with respect to the baseline benchmark (Figure 10 A). The rest of the timings remained unaffected (Figure 10 B-F). Although the absolute total benchmark time is much improved (Figure 11 A), the scaling remains challenging, with a maximum observed speedup of 12x for the contiguous file (Figure 11 B). The $N = 112$ H5MD-contiguous run's I/O time was 47s (Fig. 10 A). When compared to the 4623s baseline serial time, this is a 98x speedup. Similarly, the H5MD-chunked 4 node run resulted in an I/O time of 83s, which is a 56x speedup when compared to baseline serial performance. Therefore, the boost in performance seen by loading the H5MD-default trajectory into memory rather than iterating frame by frame is indeed most likely due to the original file's chunk layout. This emphasizes the point that one

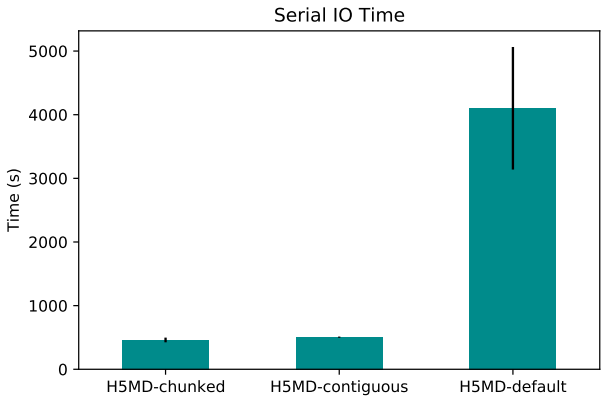

Fig. 8: *Serial I/O time for H5MD-default, H5MD-contiguous, and H5MD-chunked data files. Each file contained the same data (113 GiB, 90100 frames) but was written with a different HDF5 chunk arrangement, as outlined in Table 2. Each bar represents the mean of 5 repeat benchmark runs, with the standard deviation shown as error bars.*

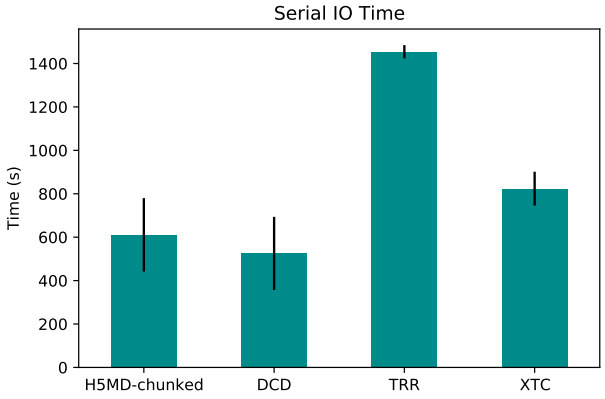

Fig. 9: *Comparison of serial I/O time for various popular MD file formats. All files contain the same amount of data (90100 frames). Each bar represents the mean of 10 repeat benchmark runs, with the standard deviation shown as error bars.*

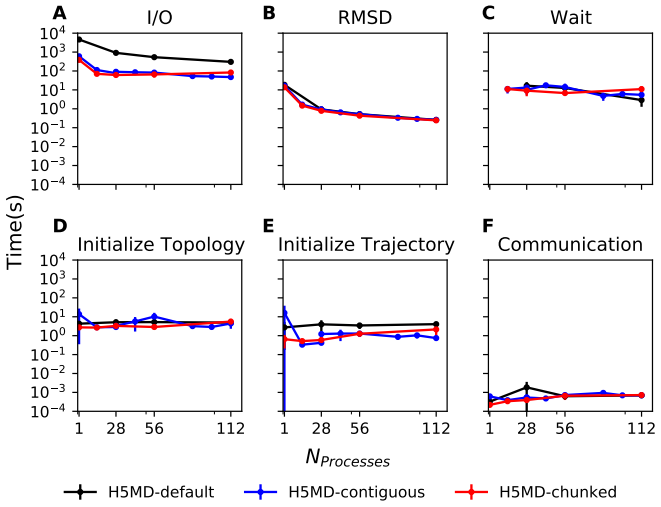

Fig. 10: *Benchmark timings breakdown on ASU Agave for the three chunk arrangements tested. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112.* ***H5MD-default*** *was auto-chunked by* `h5py`. ***H5MD-contiguous*** *was written with no chunking applied, and* ***H5MD-chunked*** *was written with a chunk shape of* `(1, n atoms, 3)`. *The trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

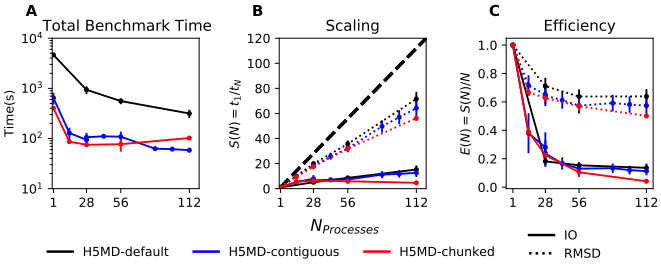

Fig. 11: *Strong scaling I/O performance of the RMSD analysis task with various chunk layouts tested on ASU Agave. $N_{processes}$ ranged from 1 core, to 4 full nodes, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

may garner substantial I/O improvements if one thinks carefully not only about how their algorithm accesses the file, but also how the file is actually stored on disk. The relationship between layout on disk and disk access pattern is crucial for optimized I/O. Furthermore, as the auto-chunked layout of the `H5MD-default` file scattered data from a single time step across multiple chunks, it is very likely that these chunks themselves were also scattered across stripes. In this case, multiple processes are still attempting to read from the same chunk which would nullify any beneficial effect striping has on file contention. We would have liked to further test the effects of striping with a proper chunk layout, but our XSEDE allocation expired.

Effects of HDF5 GZIP Compression on File I/O

HDF5 files also offer the ability to compress the files. With our writer, users are easily able to apply any of the compression settings allowed by HDF5. To see how compression affected parallel I/O, we tested HDF5's gzip compression with a minimum setting of 1 and a maximum setting of 9. In the serial 1 process case, we found that I/O performance is slightly hampered, with I/O times approximately 4x longer with compression applied (Figure 13 A) This is expected as you are giving up disk space for the time it takes to decompress the file, as is seen in the highly compressed XTC format (Fig. 9). However, at increasing number of processes ($N > 28$), we found that this difference disappears (Figure 13 A and Figure 12 A). This shows a clear benefit of applying gzip compression to a chunked HDF5 file for parallel analysis tasks, as the compressed file is ~2/3 the size of the original. Additionally we found speedups of up to 36x on 2 full nodes for the compressed data file benchmarks (Figure 13 B), although we recognize this number is slightly inflated due to the slower serial I/O time. From this data we can safely assume that H5MD files can be compressed without fear of losing parallel I/O performance, which is a nice boon in the age of terabyte sized trajectory files.

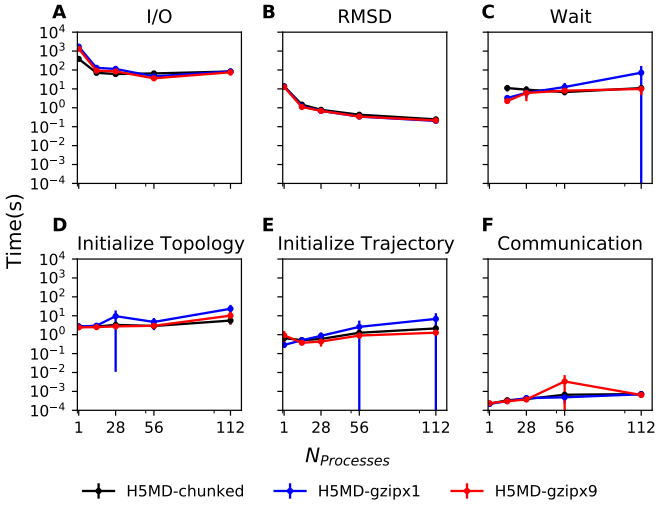

Fig. 12: *Benchmark timings breakdown on ASU Agave for the minimum gzip compression 1 and maximum gzip compression 9. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112. The trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

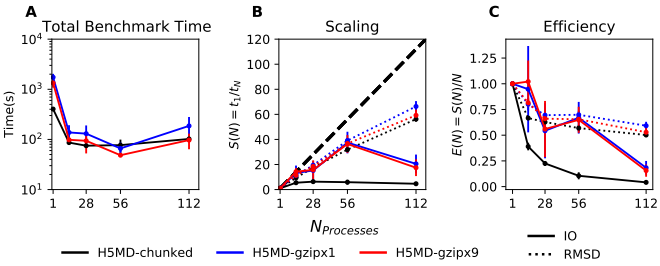

Fig. 13: *Strong scaling I/O performance of the RMSD analysis task with minimum and maximum gzip compression applied. $N_{processes}$ ranged from 1 core, to 4 full nodes, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

Conclusions

The growing size of trajectory files demands parallelization of trajectory analysis. However, file I/O has become a bottleneck in the workflow of analyzing simulation trajectories. Our implementation of an HDF5-based file format trajectory reader in MDAnalysis can perform parallel MPI I/O, and our benchmarks on various national HPC environments show that speed-ups on the order of 20x for 48 cores are attainable. Scaling up to achieve higher parallel data ingestion rates remains challenging, so we developed several algorithmic optimizations in our analysis workflows that lead to improvements in I/O times. The results from these optimization attempts led us to find that the our original data file that was auto-chunked by h5py's chunking algorithm had an incredibly inefficient chunk layout of the original file. With a custom, optimized chunk layout and gzip compression, we found maximum scaling of 36x on 2 full nodes on Agave. In terms of speedup with respect to the file chunked automatically, our properly chunked file led to I/O time speedups of 98x at 112 cores on Agave, which means carefully thinking not only about how your file is accessed, but also how the file is stored on disk can result in a reduction of analysis time from 4623 to 47 seconds. To garner further improvements in parallel I/O performance, a more sophisticated I/O pattern may be required, such as two-phase MPI I/O or carefully synchronizing chunk sizes with Lustre stripes. The addition of the HDF5 reader provides a foundation for the development of parallel trajectory analysis with MPI and the MDAnalysis package.

Acknowledgments

The authors thank Dr. Pierre de Buyl for advice on the implementation of the h5md format reading code and acknowledge Gil Speyer and Jason Yalim from the Research Computing Core Facilities at Arizona State University for support with the Agave cluster and BeeGFS. This work was supported by the National Science Foundation through a REU supplement to award ACI1443054 and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. The SDSC Comet computer at the San Diego Supercomputer Center and PSC Bridges computer at the Pittsburgh Supercomputing Center were used under allocation TG-MCB130177. The authors acknowledge Research Computing at Arizona State University for providing HPC and storage resources that contributed to the research results reported within this paper.

References

[BFJ18] Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Convergence of data generation and analysis in the biomolecular simulation community. In *Online Resource for Big Data and Extreme-Scale Computing Workshop*, page 4, November 2018. URL: https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/Beckstein-Fox-Jha_BDEC2_WP_0.pdf.

[CDG+21] Lorenzo Casalino, Abigail C Dommer, Zied Gaieb, Emilia P Barros, Terra Sztain, Surl-Hee Ahn, Anda Trifan, Alexander Brace, Anthony T Bogetti, Austin Clyde, Heng Ma, Hyungro Lee, Matteo Turilli, Syma Khalid, Lillian T Chong, Carlos Simmerling, David J Hardy, Julio DC Maia, James C Phillips, Thorsten Kurth, Abraham C Stern, Lei Huang, John D McCalpin, Mahidhar Tatineni, Tom Gibbs, John E Stone, Shantenu Jha, Arvind Ramanathan, and Rommie E Amaro. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. *The International Journal of High Performance Computing Applications*, page 10943420211006452, April 2021. Publisher: SAGE Publications Ltd STM. URL: https://doi.org/10.1177/10943420211006452, doi:10.1177/10943420211006452.

[Col14] Andrew Collette. Python and hdf5. In Meghan Blanchette and Rachel Roumeliotis, editors, *Python and HDF5*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2014.

[CR15] T. Cheatham and D. Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering*, 17(2):30–39, 2015. doi:10.1109/MCSE.2015.7.

[dBCH14] Pierre de Buyl, Peter H. Colberg, and Felix Höfling. H5MD: A structured, efficient, and portable file format for molecular data. *Computer Physics Communications*, 185(6):1546 – 1553, 2014. doi:10.1016/j.cpc.2014.01.018.

[DDG+12] Ron O Dror, Robert M Dirks, J P Grossman, Huafeng Xu, and David E Shaw. Biomolecular simulation: a computational microscope for molecular biology. *Annu Rev Biophys*, 41:429–52, 2012. doi:10.1146/annurev-biophys-042910-155245.

[DPKC11] Lisandro D. Dalcín, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools. doi:10.1016/j.advwatres.2011.04.013.

[FQC+19] Geoffrey Fox, Judy Qiu, David Crandall, Gregor von Laszewski, Oliver Beckstein, John Paden, Ioannis Paraskevakos, Shantenu Jha, Fusheng Wang, Madhav Marathe, Anil Vullikanti, and III Cheatham, Thomas E. Contributions to high-performance big data computing. In L. Grandinetti, G. R. Joubert, K. Michielsen, S. L. Mirtaheri, M. Taufer, and R Yokota, editors, *Future Trends of HPC in a Disruptive Scenario*, volume 34 of *Advances in Parallel Computing*, pages 34–81. IOS Press, 2019. doi:10.3233/APC190005.

[GLB+16] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, David L Dotson, Jan Domański, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98–105, Austin, TX, 2016. SciPy. doi:10.25080/Majora-629e541a-00e.

[HBD+19] David J. Huggins, Philip C. Biggin, Marc A. Dämgen, Jonathan W. Essex, Sarah A. Harris, Richard H. Henchman, Syma Khalid, Antonija Kuzmanic, Charles A. Laughton, Julien Michel, Adrian J. Mulholland, Edina Rosta, Mark S. P. Sansom, and Marc W. van der Kamp. Biomolecular simulations: From dynamics and mechanisms to computational assays of biological activity. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 9(3):e1393, 2019. doi:10.1002/wcms.1393.

[HMvdW+20] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, 09 2020. doi:10.1038/s41586-020-2649-2.

[How10] Mark Howison. Tuning HDF5 for Lustre File Systems. September 2010. URL: https://escholarship.org/uc/item/46r9d86r.

[KPF+20] Mahzad Khoshlessan, Ioannis Paraskevakos, Geoffrey C. Fox, Shantenu Jha, and Oliver Beckstein. Parallel performance of molecular dynamics trajectory analysis. *Concurrency and Computation: Practice and Experience*, 32:e5789, 2020. doi:10.1002/cpe.5789.

[LRFK+21] Maria Lopez-Redondo, Shujie Fan, Akiko Koide, Shohei Koide, Oliver Beckstein, and David L. Stokes. Zinc binding alters the conformational dynamics and drives the transport cycle of the cation diffusion facilitator YiiP. *Journal of General Physiology*, 153(8), July 2021. URL: https://doi.org/10.1085/jgp.202112873, doi:10.1085/jgp.202112873.

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

[MM14] Cameron Mura and Charles E. McAnany. An introduction to biomolecular simulations and docking. *Molecular Simulation*, 40(10-11):732–764, 2014. doi:10.1080/08927022.2014.935372.

[Oro14] Modesto Orozco. A theoretical view of protein dynamics. *Chem. Soc. Rev.*, 43:5051–5066, 2014. doi:10.1039/C3CS60474H.

[PLK+18] Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, Goerge Chantzialexiou, Thomas E. Cheatham, Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Task-parallel analysis of molecular dynamics trajectories. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*, page Article No. 49, New York, NY, USA, August 13–16 2018. Association for Computing Machinery, ACM. doi:10.1145/3225058.3225128.

[SFMLIP+19] Shujie Fan, Max Linke, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 134 – 142, Austin, TX, 2019. SciPy. doi:10.25080/Majora-7ddc1dd1-013.

[TCD+14] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, Sept.-Oct. 2014. doi:10.1109/MCSE.2014.80.

[The05] Douglas L Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallogr A*, 61(Pt 4):478–80, Jul 2005. doi:10.1107/S0108767305015266.

[Wic11] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. doi:10.18637/jss.v040.i01.

Natural Language Processing with Pandas DataFrames

Frederick Reiss[‡*], Bryan Cutler[§], Zachary Eichenberger[¶‡]

Abstract—Most areas of Python data science have standardized on using Pandas DataFrames for representing and manipulating structured data in memory. Natural Language Processing (NLP), not so much.

We believe that Pandas has the potential to serve as a universal data structure for NLP data. DataFrames could make every phase of NLP easier, from creating new models, to evaluating their effectiveness, to building applications that integrate those models. However, Pandas currently lacks important data types and operations for representing and manipulating crucial types of data in many of these NLP tasks.

This paper describes *Text Extensions for Pandas*, a library of extensions to Pandas that make it possible to build end-to-end NLP applications while representing all of the applications' internal data with DataFrames. We leverage the extension points built into Pandas library to add new data types, and we provide important NLP-specfific operations over these data types and and integrations with popular NLP libraries and data formats.

Index Terms—natural language processing, Pandas, DataFrames

Background and Motivation

This paper describes our work on applying general purpose data analysis tools from the Python data science stack to Natural Language Processing (NLP) applications. This work is motivated by our experiences working on NLP products from IBM's *Watson* portfolio, including IBM Watson Natural Language Understanding [Intb] and IBM Watson Discovery [Inta].

These products include many NLP components, such as state-of-the-art machine learning models, rule engines for subject matter experts to write business rules, and user interfaces for displaying model results. However, the bulk of the development work on these products involves not the core NLP components, but data manipulation tasks, such as converting between the output formats of different models, manipulating training data, analyzing the outputs of models for correctness, and serializing data for transfer across programming language and machine boundaries.

Although the raw input to our NLP algorithms is text in a natural language, most of the code in our NLP systems operates over machine data. Examples of this machine data include:

- Relational tables of training data in formats like CoNLL-U [NdMG+20]
- Model outputs formatted as tables for comparison against training data
- Arrays of dense tensors that represent BERT embeddings [DCLT19]
- Graphs that represent dependency-based parse trees
- Relational tables that represent document structure

This focus on data manipulation tasks instead of core AI algorithms is not unique to IBM, or indeed to NLP [SHG+15]. However, NLP is unique in the quantity of redundant data structures and low-level algorithms that different systems reimplement over and over again. One can see this trend clearly in open source NLP libraries, where free access to internal code also exposes the internal data structures. Each of the major NLP libraries implements its own custom data structures for basic NLP concepts.

Consider the concept of a *span*: a region of a document, usually expressed as a range of characters or tokens. NLP systems use spans to represent the locations of information they extract from text. This information includes tokens, named entities, arguments to semantic role labeling predicates, and many others.

Here is how some popular Python NLP libraries represent spans:

- *spaCy* [HMVLB20] has a Python class named `Span` that represents a range of tokens. The locations of these tokens are stored inside the class `Doc`. The `__getitem__` method of `Doc` returns instances of the class `Token`, which encodes the location of the token as a beginning character offset and a length in characters [Exp21].
- *Stanza* [QZZ+20] has a Python class also named `Span` that represents a range of *characters*. Information about the tokens that are contained within the character range is stored in the `tokens` property of the `Span` as objects of type `Token` [Mai21a]. These classes, `Span` and `Token`, are different from the `spaCy` classes with the same names.
- `nltk` [LB02] models text as a Python list. Depending on the stage of processing, the elements of the list can be Python strings or tuples. Spans over tokens are represented by slices of the list, and information about character locations is generally not available [BKL09].
- `transformers` [WDS+20] does not generally model spans; instead it leaves that choice up to the user. One exception to this policy is the library's `TokenClassificationPipeline` class, which has a method `group_entities` that returns a Python dictionary for each entity. The fields `start` and `end` in

* Corresponding author: frreiss@us.ibm.com
‡ IBM Research
§ IBM
¶ University of Michigan

this dictionary hold the span of the entity, measured in characters [Hug21].
- *TensorFlow Text* [Mai21b] represents lists of spans as either a pair of one-dimensional tensors (for tokenization) or as a single two-dimensional tensor (for span comparison operations). The elements of the tensors can represent byte, character, or token offsets. Users need to track which type of offset is stored in a given tensor [Mai21c].

All of these representations are incompatible with each other. Users who want to use any two of these libraries together will need to write code to convert between their outputs. Users are also left to invent their own algorithms for even the most basic operations over spans, including serializing them, finding their covered text, determining whether two spans overlap, and finding matches between two sets of spans.

The redundancy that these libraries display at the level of individual spans is pervasive across all the more complex structures that they extract from text. Both users and library developers spend considerable amounts of time reading the documentation for these different data structures, writing code to convert between them, and reimplementing basic operations over them.

An Alternative Approach

The Python data science community has developed effective tools for managing and analyzing data in memory, chief among them being the DataFrame library *Pandas* [pdt21b]. Could we use these general-purpose tools instead of continually reinventing data structures and algorithms for basic NLP tasks?

We prototyped some use cases and quickly discovered that NLP-related data involves domain-specific concepts; and some of these concepts are inconvenient to express in Pandas. For example, the *span* concept that we described in the previous section is a crucial part of many applications. The closest analog to a span in Pandas' data model is the `interval` type, which represents an interval using a pair of numbers. When we prototyped some common NLP applications using `interval` to represent spans, we needed additional code and data structures to track the relationships between intervals and target strings; as well as between spans and different tokenizations. We also needed code to distinguish between intervals measured in characters and in tokens. All of this additional code negated much of the benefit of the general-purpose tool.

To reduce the amount of code that users would need to write, we started working on extensions to Pandas to better represent NLP-specific data and to support key operations over that data. We call the library that we eventually developed *Text Extensions for Pandas*.

Extending Pandas

Text Extensions for Pandas includes three types of extensions:

- NLP-specific **data types (dtypes)** for Pandas DataFrames
- NLP-specific **operations** over these new data types
- **Integrations** between Pandas and common NLP libraries

Pandas includes APIs for library developers to add new data types to Pandas, and we used these facilities to implement the NLP-specific data types in Text Extensions for Pandas.

The core component of the Pandas extension type system is the *extension array*. The Python class `pandas.api.extensions.ExtensionArray` defines key operations for a columnar array object that backs a Pandas `Series` [pdt21a]. Classes that extend `ExtensionArray` and implement a relatively short list of required operations can serve as the backing stores for Pandas `Series` objects while supporting nearly all the operations that Pandas built-in types support, including filtering, slicing, aggregation, and binary I/O.

Indeed, many of the newer built-in types in Pandas, such as the `interval` and `categorical`, are implemented as subclasses of `ExtensionArray`. Text Extensions for Pandas includes three different extension types based on this API. The first two extension types are for spans with character- and token-based offsets, respectively. The third extension type that we add represents tensors.

Spans

We implement character-based spans with a Python class called `SpanArray`, which derives from Pandas' `ExtensionArray` base class. A `SpanArray` object represents a column of span data, and it stores this data internally using three NumPy [HMvdWea20] arrays, plus a shared reference to the underlying text.

The three arrays that represent a column of span data consist of arrays of begin and end offsets (in characters), plus a third array of indices into a table of unique target strings. The `SpanArray` object also stores a shared reference to this table of strings.

The string table is necessary because a Pandas Series can contain spans over many target strings. The spans in the Series might come from multiple documents, or they may come from multiple fields of the same document. Users need to be able to perform operations over the containing DataFrames without performing many string equality checks or creating many copies of the text of each document. Representing the target text of each span as an index into the table allows us to quickly check whether two spans are over the same string. The string table also allows the `SpanArray` class to track exactly which unique strings the array's spans cover. Keeping track of this set of strings is important for efficient serialization, as well as for efficiently appending one `SpanArray` to another. As an additional optimization, slicing and filtering operations over a `SpanArray` do not modify the string table; a slice of an array will share the same table as the original array.

In addition to spans with character offsets, we also support spans whose begin and end offsets are measured in tokens. Most machine learning models and rule engines for NLP do not operate over sequences of characters but over sequences of *tokens*—ranges of characters that correspond to elements like words, syllables, or punctuation marks. Character-based spans are useful for comparing, visualizing, and combining the outputs of multiple models, because those models may use different tokenizations internally. When analyzing the inputs and outputs of a single model (or rule set, in the case of a rule-based NLP system), tokens are a more appropriate unit for the begin and end offsets of spans. Representing spans with token offsets allows for operations like computing token distances between spans and can prevent errors that could lead to spans not starting or ending on a token boundary. The loss functions used to train most NLP models also tend to operate over tokens.

There can be multiple different tokenizations of the same document, even within a single application. When storing token-based span offsets, it is important to retain information about

which tokenization of which document each token offset corresponds to. The `TokenSpanArray` class represents each distinct tokenization of a document with an instance of `SpanArray` containing the locations of the tokens. The representation of the token-based spans themselves consists of three NumPy arrays, holding begin and end offsets (in tokens) and a pointer to the `SpanArray` containing the token offsets.

Although it stores the locations of spans as token offsets, the `TokenSpanArray` class can generate character-based begin and offsets on demand from its internal tables of token locations. This facility allows `TokenSpanArray` to be used in any code that works over instances of `SpanArray`. For example, code that detects pairs of overlapping spans can easily work over arbitrary combinations of token- and character-based spans, which is useful when merging the outputs of models that represent span offsets differently.

The internal structure of our `SpanArray` and `TokenSpanArray` extension arrays allows for efficient vectorized implementations of common Pandas operations like slicing, filtering, and aggregation. Slicing operations over a `SpanArray` produce a new `SpanArray` with views of the original `SpanArray` object's internal NumPy arrays, avoiding unnecessary copying of span data.

Tensors

Tensors—dense n-dimensional arrays—are another common concept in modern NLP. The deep learning models that drive much of state-of-the-art NLP today take tensors as inputs and outputs and operate internally over other tensors. Embeddings—data structures that encode information about a block of text as a dense vector amenable to analysis with algorithms that expect dense input—are a key part of many NLP algorithms and can be efficiently represented with tensors. Tensors are also useful for more traditional types of NLP data, such as n-grams and one-hot-encoded feature vectors.

Our `TensorArray` extension array class represents a Pandas Series where each element is a tensor. Internally, we represent the entire Series' data as a single dense NumPy array. The TensorArray class translates Pandas array operations to vectorized operations over the underlying NumPy array. Because CPython [cd21], the most common runtime for Python, uses an intepreter to run Python code, these vectorized operations are much more efficient than iterating over a list of tensors.

Since the individual data items in a `TensorArray` are actually slices of a larger NumPy array, our tensor data type integrates seamlessly with third party libraries that accept NumPy arrays. For example, Figure 1 shows how our tensor data type works with the `matplotlib` [Hun07] plotting library in a Jupyter notebook.

Some libraries, notably `xarray` [HH17], provide Pandas-like dataframes specialized for numeric tensor or array data. These libraries are useful for cases where dataframes consist almost entirely of tensor data. Our `TensorArray` extension type is a complementary alternative for applications where the data is a mixture of tensors, spans, and built-in Pandas data types with a wide variety of different schemas. For example, figure 2 shows an example of a DataFrame that mixes spans, tensors, and Pandas categorical types to store features of the tokens in a document. For applications that need this kind of mixture of data, our tensor type allows users to leverage Pandas' collection of built-in operations and third-party visualizations, while still operating efficiently over tensor-valued data series.

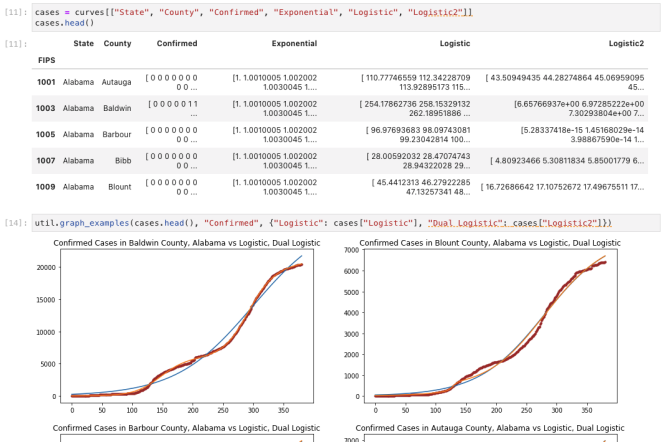

Fig. 1: *Example of using our tensor data type to store a time series while visualizing those time series with the `matplotlib` [Hun07] library in a Jupyter notebook. In the top half of the window is a DataFrame where each cell of the rightmost four columns contains an entire time series of COVID-19 case data as a tensor. The bottom half of the screen shows the results of plotting these tensors directly out of the DataFrame. This example notebook is available at https://github.com/CODAIT/covid-notebooks/blob/master/notebooks/analyze_fit_us_data.ipynb.*

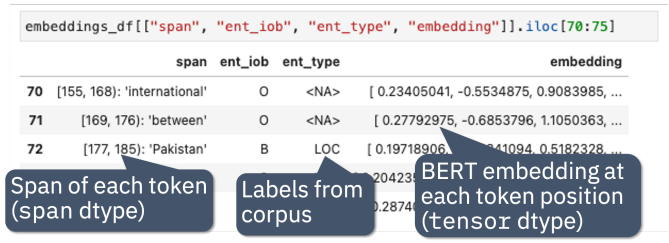

Fig. 2: *Slice of a DataFrame of information about tokens constructed with our library's integration with the `transformers` library for masked language models. Each row of the DataFrame represents a token in the document. The leftmost column uses our span extension type to store the position of the token. The rightmost column stores a BERT embedding at that token position. The columns in between hold token metadata that was created by aligning the corpus' original tokenization with the language model's tokenization, then propagating the corpus labels between pairs of aligned tokens. The notebook in which this example appears (available at https://github.com/CODAIT/text-extensions-for-pandas/blob/master/notebooks/Model_Training_with_BERT.ipynb) shows how to use this DataFrame as the input for training a named entity recognition model with the `sklearn` library.*

Serialization

Many areas of modern NLP involve large collections of documents, and common NLP operations can expand the size of this data by orders of magnitude. Pandas includes facilities for efficient serialization of Pandas data types using Apache Arrow [Com21]. Text Extensions for Pandas uses this support to convert data from the library's extension types into Arrow format for efficient storage and transfer.

Efficient binary I/O can make reading and writing NLP corpora orders of magnitude faster. Figure 3 compares the amount of time required to read the training fold of the CoNLL-2003 corpus [TKSDM03] from a local filesystem when the corpus is stored in three different formats. Reading the corpus with Pandas and the

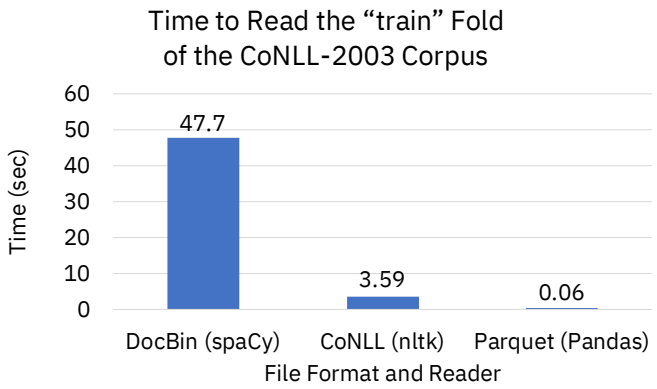

Fig. 3: *Comparison of the amount of time required to read the training fold of the CoNLL-2003 named entity recognition corpus into memory, when the corpus is stored in three different file formats. Binary I/O with Pandas and the Apache Parquet file format is 2-3 orders of magnitude faster than the other file formats tested.*

Apache Parquet binary file format is 60 times faster than reading the original CoNLL-format text file with `nltk` and 800 times faster than reading the corpus in DocBin format with spaCy.

Text Extensions for Pandas also supports reading files in the text-based formats known as CoNLL and CoNLL-U. Many benchmark datasets for NLP are released in these formats. Text Extensions for Pandas can convert these files into DataFrames with one line per token, using our span extension type to store the location of a given token and the location of the sentence that contains the token.

Spanner Algebra

In addition to representing span data, NLP applications need to filter, transform, and aggregate this data, often in ways that are unique to NLP.

The *document spanners* formalism [FKRV15] extends the relational algebra with additional operations to cover a wide gamut of critical NLP operations.

Since it is an extension of the relational algebra, much of document spanners can already be expressed with Pandas core operations. We have implemented several of the remaining parts of document spanners as operations over Pandas Series of data type Span.

Specifically, we have NLP-specific *join* operations (sometimes referred to as "merge") for identifying matching pairs of spans from two input sets, where the spans in a matching pair have an overlap, containment, or adjacency relationship. These join operations are crucial for combining the results of multiple NLP models, and they also play a role in rule-based business logic. For example, a domain expert might need to find out matches of one model that overlap with matches of a different model. If the output spans are in the "span" columns of two DataFrames, `model_1_out` and `model_2_out`, then the user can find all such matching pairs by running the following line of code:

```
import text_extensions_for_pandas as tp

# Find output spans of model 1 that contain output
# spans of model 2.
# This expression returns a DataFrame with two
# columns, span_1 and span_2, both of type span.
span_pairs = tp.spanner.contain_join(
    model_1_out["span"], model_2_out["span"],
    "span_1", "span_2")
```

We include two implementations of the *extract* operator, which produces a set of spans over the current document that satisfy a constraint. Our current implementations of *extract* support extracting the set of spans that match a regular expression or a gazetteer (dictionary).

We include a version of the *consolidate* operator, which takes as input a set of spans and removes overlap among the spans by applying a consolidation policy. This operator is useful for business logic that combines the results of multiple models and/or extraction rules as well as for resolving ambiguity when a single model produces overlapping spans in its output.

Other Span Operations

We support span operations that are not part of the document spanners formalism but are important for key NLP tasks. These operations include:

- aligning spans based on one tokenization of the document to a different tokenization
- *lemmatizing* spans—that is, converting the text that the span covers to a normalized form
- converting sequences of tokens tagged with inside-outside-beginning (IOB) tags [RM95] into spans of entities, and vice versa.

Jupyter Notebook Integration

Jupyter notebooks have built-in facilities for displaying Pandas DataFrames. Our extensions to Pandas also work with these facilities. If the last line of a notebook cell returns a DataFrame containing span and tensor data, then Jupyter will display an HTML representation of the DataFrame, including cells that contain our extension types. Figure 2 shows how a DataFrame containing a column of spans and a column of tensors renders as HTML when shown in a Juypter notebook.

Other Python development tools, including Visual Studio Code, PyCharm, and Google Colab, use extended versions of the Jupyter DataFrame display facilities to show DataFrames in their own user interfaces. Our extension types also work with these interfaces.

There is also an ecosystem of interactive libraries for exploring and visualizing Pandas DataFrames. Examples of such libraries include D-Tale [dt21a], Qgrid [dt21b], and the Spyder [Con21] Variable Explorer. These libraries also work with our extension types. Figure 4 shows an example of using Text Extensions for Pandas to display span data with the *D-Tale* interactive data analysis tool [dt21a].

Because our extension types for tensors use NumPy's *ndarray* type for individual cell values, these extension types work with many tools that accept NumPy arrays. Figure 1 shows an example of storing time series in the cells of a DataFrame and plotting those time series directly out of the DataFrame using the graphics library `matplotlib` in a Jupyter notebook.

It is often useful to visualize spans in the context of the source text. We use Jupyter's built-in application programming interface (API) for HTML rendering to facilitate this kind of visualization. If the last expression in a notebook cell returns a *SpanArray* or *TokenSpanArray* object, then Jupyter will automatically display the spans in the context of the target text, as shown in Figure 5.

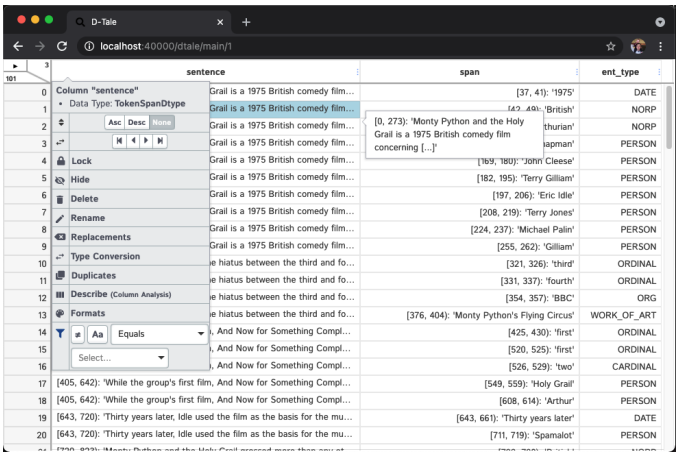

Fig. 4: *Displaying a DataFrame containing span data in the* D-Tale *interactive visualizer [dt21a]. Our extension types for NLP work with third-party libraries without requiring any changes to those libraries.*

```
[26]: mask = pronouns_by_sentence["sentence"].map(lambda s: s.covered_text).str.contains("Arthur")
      pronouns_by_sentence["span"][mask].values
```

[26]:

	begin	end	covered_text
0	27	30	his
1	455	457	he
2	490	492	it
3	516	520	they
4	572	576	them

In AD 932, King Arthur and his squire, Patsy, travel throughout Britain searching for men to join the Knights of the Round Table. Along the way, he recruits Sir Bedevere the Wise, Sir Lancelot the Brave, Sir Galahad the Pure, Sir Robin the Not-Quite-So-Brave-as-Sir-Lancelot, and Sir Not-Appearing-in-this-Film, along with their squires and Robin's troubadours. Arthur leads the men to Camelot, but upon further consideration (thanks to a musical number) he decides not to go there because it is "a silly place". As they turn away, God (an image of W. G. Grace) speaks to them and gives Arthur the task of finding the Holy Grail.

Fig. 5: *Displaying the contents of a Pandas Series of span data in the context of the target document, using the integration between Text Extensions for Pandas and Jupyter's APIs for HTML display. The spans shown in this example represent all pronouns in sentences that contain the name "Arthur". We generated this set by cross-referencing the outputs of two models using Pandas operations. This notebook can be found at https://github.com/CODAIT/text-extensions-for-pandas/blob/master/notebooks/Analyze_Text.ipynb.*

Taken together with JupyterLab's ability to display multiple widgets and views of the same notebook, these facilities allow users to visualize NLP data from several perspectives at once, as shown in Figure 11.

NLP Library Integrations

Text Extensions for Pandas provides facilities for transforming the outputs of several common NLP libraries into Pandas DataFrames to represent NLP concepts.

spaCy

spaCy [HMVLB20] is a Python library that provides a suite of NLP models intended for production use. Users of spaCy access most of the library's functionality through spaCy *language models*, Python objects that encapsulate a pipeline of rule-based and machine learning models. A spaCy language model takes natural language text as input and extracts features such as parts of speech, named entities, and dependency relationships from the text. These features are useful in various downstream NLP tasks.

Our spaCy integration converts the output of a spaCy language model into a DataFrame of token information. Figure 6 shows an

```
[19]: doc_text = response["analyzed_text"]
      spacy_language_model = spacy.load("en_core_web_sm")
      token_features = tp.io.spacy.make_tokens_and_features(doc_text, spacy_language_model)
      token_features
```

[19]:

	id	span	lemma	pos	tag	dep	head	shape	ent_iob	ent_type	is_alpha	is_stop	sentence
0	0	[0, 2): 'In'	in	ADP	IN	prep	12	Xx	O		True	True	[0, 129): 'In AD 932, King Arthur and his squi...
1	1	[3, 5): 'AD'	ad	NOUN	NN	pobj	0	XX	B	DATE	True	False	[0, 129): 'In AD 932, King Arthur and his squi...
2	2	[6, 9): '932'	932	NUM	CD	nummod	1	ddd	I	DATE	False	False	[0, 129): 'In AD 932, King Arthur and his squi...
3	3	[9, 10): ','	,	PUNCT	,	punct	12	,	O		False	False	[0, 129): 'In AD 932, King Arthur and his squi...
4	4	[11, 15): 'King'	King	PROPN	NNP	compound	5	Xxxx	B	ORG	True	False	[0, 129): 'In AD 932, King Arthur and his squi...
...	...	...	...	...	...	...	...	...	...	...	...	...	...
142	142	[606, 613): 'finding'	find	VERB	VBG	pcomp	141	xxxx	O		True	False	[513, 629): 'As they turn away, God (an image ...
143	143	[614, 617): 'the'	the	DET	DT	det	145	xxx	B	WORK_OF_ART	True	True	[513, 629): 'As they turn away, God (an image ...
144	144	[618, 622): 'Holy'	Holy	PROPN	NNP	compound	145	Xxxx	I	WORK_OF_ART	True	False	[513, 629): 'As they turn away, God (an image ...
145	145	[623, 628): 'Grail'	Grail	PROPN	NNP	dobj	142	Xxxxx	I	WORK_OF_ART	True	False	[513, 629): 'As they turn away, God (an image ...
146	146	[628, 629): '.'	.	PUNCT	.	punct	133	.	O		False	False	[513, 629): 'As they turn away, God (an image ...

147 rows × 13 columns

Fig. 6: *Example of converting the output of a spaCy language model. Each row of the DataFrame holds information about a single token, including the span of the token and the span of the containing sentence. The code for this example is available at https://github.com/CODAIT/text-extensions-for-pandas/blob/master/notebooks/Integrate_NLP_Libraries.ipynb.*

example of using this integration to process the first paragraph of the Wikipedia article for the film *Monty Python and the Holy Grail*.

Converting from spaCy's internal representation to DataFrames allows usage of Pandas operations to analyze and transform the outputs of the language model. For example, users can use Pandas' filtering, grouping, and aggregation to count the number of nouns in each sentence:

```
# Filter tokens to those that are tagged as nouns
nouns = tokens[tokens["pos"] == "NOUN"]

# Compute the number of nouns in each sentence
nouns.groupby("sentence").size() \
    .to_frame(name="num_nouns")
```

Or they could use our span-specific join operations and Pandas' *merge* function to match all pronouns in the document with the person entities that are in the same sentence:

```
import text_extensions_for_pandas as tp

# Find person names
entities = tp.io.conll.iob_to_spans(tokens)
person_names = entities[
    entities["ent_type"] == "PERSON"]["span"]

# Find all pronouns
pronouns = tokens[tokens["tag"] == "PRP"] \
    [["span", "sentence"]]

# Find all sentences
sentences = tokens[["sentence"]].drop_duplicates() \
    ["sentence"]

# Match names and pronouns in the same sentence
pronoun_person_pairs = (
    pronouns.rename(columns={"span": "prounoun"})
        .merge(tp.spanner.contain_join(
            sentences, person_names,
            "sentence", "person")))
```

We also support using spaCy's *DisplaCy* visualization library to display dependency parse trees stored in DataFrames. Users can filter the output of the language model using Pandas operations, then display the resulting subgraph of the parse tree in a Jupyter notebook. This display facility will work with any DataFrame that encodes a dependency parse as a Pandas Series of token spans, token IDs, and head IDs.

transformers

`transformers` [WDS+20] is a library that provides implementations of many state of the art masked language models

such as BERT [DCLT19] and RoBERTa [LOG+19]. In addition to the language models themselves, `transformers` includes dedicated tokenizers for these models, most of which use subword tokenizers like *SentencePiece* [KR18] to improve accuracy.

Text Extensions for Pandas can transform two types of outputs from the `transformers` library for masked language models into Pandas DataFrames. We can convert the output of the library's tokenizers into DataFrames of token metadata, including spans marking the locations of each token.

Our tensor data type can also represent embeddings from the encoder stage of a `transformers` language model. Since the language models in `transformers` have a limited sequence length, we also include utility functions for dividing large DataFrames of token information into fixed-size windows, generating embeddings for each window, and concatenating the resulting embeddings to produce a new column for the original DataFrame. Figure 2 shows a DataFrame of token features that includes both a span column with token location and a tensor column with embeddings at each token position.

IBM Watson Natural Language Understanding

Watson Natural Language Understanding [Intb] is an API that provides access to prebuilt NLP models for common tasks across a wide variety of natural languages. Users can use these APIs to process several thousands documents per month for free, with paid tiers of the service available for higher data rates.

Our Pandas integration with Watson Natural Language Understanding can translate the outputs of all of Watson Natural Language Understanding's information extraction models into Pandas DataFrames. The supported models are:

- *syntax*, which performs syntax analysis tasks like tokenization, lemmatization, and part of speech tagging.
- *entities*, which identifies mentions of named entities such as persons, organizations, and locations.
- *keywords*, which identifies instances of a user-configurable set of keywords as well as information about the sentiment that the document expresses towards each keyword.
- *semantic_roles*, which performs *semantic role labeling*, extracting subject-verb-object triples that describe events which occur in the text.
- *relations*, which identifies relationships betwen pairs of named entities.

Converting the outputs of these models to DataFrames makes building notebooks and applications that analyze these outputs much easier. For example, with two lines of Python code, users can produce a DataFrame with information about all person names that a document mentions:

```
import text_extensions_for_pandas as tp

# The variable "response" holds the JSON output
# of the Natural Language Understanding service.
# Convert to DataFrames and retrieve the DataFrame
# of entity mentions.
entities = tp.io.watson.nlu.parse_response(response) \
           ["entity_mentions"]

# Filter entity mentions down to just mentions of
# persons by name.
persons = entities[entities["type"] == "Person"]
```

Figure 7 shows the DataFrame that this code produces when run over an IBM press release.

	type	text	span	confidence
37	Person	Daniel Hernandez	[1288, 1304): 'Daniel Hernandez'	0.994301
39	Person	Curren Katz	[1838, 1849): 'Curren Katz'	0.990223
43	Person	Ritu Jyoti	[2476, 2486): 'Ritu Jyoti'	0.713109
52	Person	Tyler Allen	[4213, 4224): 'Tyler Allen'	0.964611

Fig. 7: *DataFrame of person names in a document created by converting the output of the Watson Natural Language Understanding's `entities` model to a DataFrame of entity mentions. We then used Pandas filtering operations to select the entity mentions of type "Person". The first column holds spans that tell where in the document each mention occurred. The original press release can be found at https://newsroom.ibm.com/2020-12-02-IBM-Named-a-Leader-in-the-2020-IDC-MarketScape-For-Worldwide-Advanced-Machine-Learning-Software-Platform.*

	person	url
0	[1977, 1991): 'Wendi Whitmore'	https://newsroom.ibm.com/2020-02-11-IBM-X-Forc...
0	[1281, 1292): 'Rob DiCicco'	https://newsroom.ibm.com/2020-02-18-IBM-Study-...
0	[1213, 1229): 'Christoph Herman'	https://newsroom.ibm.com/2020-02-19-IBM-Power-...
1	[2227, 2242): 'Stephen Leonard'	https://newsroom.ibm.com/2020-02-19-IBM-Power-...
0	[2289, 2297): 'Bob Lord'	https://newsroom.ibm.com/2020-02-26-2020-Call-...
...	...	...

Fig. 8: *Excerpt from DataFrame containing the names of 301 executives extracted from 191 IBM press releases. To generate this table, we first converted the outputs of Watson Natural Language Understanding's `entities` model, which finds mentions of person names, and the product's `semantic_roles` model, which extracts information about the context in which words occur. Then we used a series of standard Pandas operations, plus operations from spanner algebra, to cross-reference the outputs of the two models. Code and a full explanation of this use case can be found in the article "Market Intelligence with Pandas and IBM Watson on the IBM Data and AI blog [RC21].*

With a few additional steps, users can combine the results of multiple models to produce sophisticated document analysis pipelines. Figure 8 shows a DataFrame with the names of 301 executives extracted from 191 IBM press releases by cross-referencing the outputs of Watson Natural Language Understanding's `entities` and `semantic_roles` models. All of the analysis steps that went into producing this result were done with high-level operations from Pandas and Text Extensions for Pandas. Source code for this example is available on our blog post about this use case [RC21].

IBM Watson Discovery

IBM Watson Discovery [Inta] is a document management platform that uses intelligent search and text analytics to eliminate barriers to sharing data between teams and to retrieve information buried inside enterprise data. One of the key features of the IBM Watson Discovery product is *Table Understanding*, a document enrichment model that identifies and parses human-readable tables of data in PDF and HTML documents.

Text Extensions for Pandas can convert the output of Watson Discovery's Table Understanding enrichment into Pandas DataFrames. This facility allows users to reconstruct the contents and layout of the original table as a DataFrame, which is useful for debugging and analysis of these outputs. Figure 9 shows an

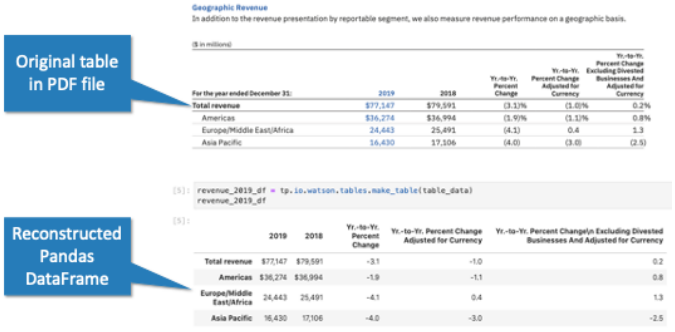

Fig. 9: *An example table from a PDF document in its original, human-readable form (left) and after using Text Extensions for Pandas to convert the output of Watson Discovery's Table Understanding enrichment into a Pandas DataFrame.*

Year	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Region												
Americas	42807.0	40184.0	42044.0	44944.0	44556.0	43249.0	41410.0	38486.0	37513.0	37479.0	36994.0	36274.0
Asia Pacific	21111.0	20710.0	23150.0	25273.0	25937.0	22923.0	20216.0	16871.0	17313.0	16970.0	17106.0	16430.0
Europe/Middle East/Africa	37020.0	32583.0	31866.0	33952.0	31775.0	31628.0	30700.0	26073.0	24769.0	24345.0	25491.0	24443.0

Fig. 10: *DataFrame containing ten years of IBM revenue broken down by geography, obtained by loading ten years of IBM annual reports int IBM Watson Discovery; converting the outputs of Watson Discovery's Table Understanding enrichment to DataFrames; then cleaning and deduplicating the resulting data using Pandas. The code that produced this result can be found at https://github.com/CODAIT/text-extensions-for-pandas/blob/master/notebooks/Understand_Tables.ipynb.*

example DataFrame from this process next to the original table in the source PDF document.

Our conversion also produces the "shredded" representation of the table as a DataFrame with one line for each cell of the original table. This data format facilitates data integration and cleaning of the extracted information. Pandas' facilities for data cleaning, filtering, and aggregation are extremely useful for turning raw information about extracted tables into clean, deduplicated data suitable to insert into a database. Figure 10 shows how, by cleaning and merging this shredded representation of a revenue table across multiple IBM annual reports, one can construct a DataFrame with ten years of revenue information broken down by geography.

Usage in Natural Language Processing Research

We are using Text Extensions for Pandas in ongoing research on semisupervised identification of errors in NLP corpora. Pandas' data analysis facilities provide a powerful substrate for cross-referencing and analyzing the outputs of NLP models in order to pinpoint potentially-incorrect labels.

One example of this type of application is work that we and several other coauthors recently published on correcting errors in the highly-cited CoNLL-2003 corpus for named entity recognition [RXC+20]. We identified over 1300 errors in the corpus and published a corrected version of the corpus. We also revisited recent results in named entity recognition using the corrected corpus.

Nearly every step of our analysis used Text Extensions for Pandas. We started by using our library's input format support to read the model results from the 16 teams in the dataset's original 2003 competition. Then we used Text Extensions for Pandas to convert these outputs from labeled tokens to DataFrames of <span, label> pairs, with one such pair for each entity mention. Using spanner algebra, we cross-referenced these entity mentions with the entity mentions to find cases where there was strong agreement among the teams' models coupled with *disagreement* with the corpus labels. A large fraction of these cases involved incorrect corpus labels.

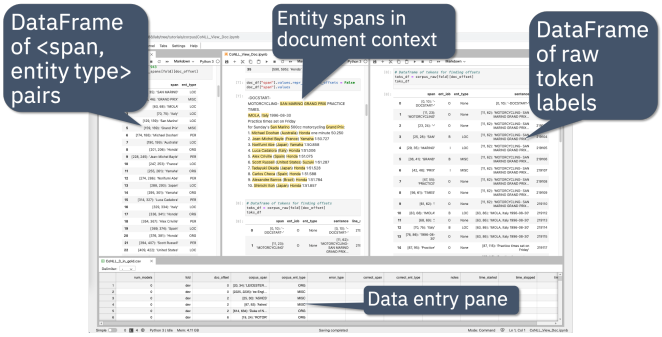

Fig. 11: *Example of using our extensions to Pandas and JupyterLab to create an ad-hoc interface for inspecting potentially incorrect labels in a named entity recognition corpus. The top three panes of this JupyterLab session display three different views of a collection of named entities for human evaluation. All of these views are driven off of Pandas DataFrames of <span, entity type> pairs. The bottom pane is where human evaluators flag incorrectly labeled entities. This Jupyter notebook is part of an in-depth tutorial available at https://github.com/CODAIT/text-extensions-for-pandas/tree/master/tutorials/corpus.*

Since we did not have model outputs for the training fold of the corpus, we used our library's integration with the `transformers` library to retokenize this part of the corpus with the BERT tokenizer. Then we used spanner algebra to match the corpus's token labels with the corresponding subword tokens from the BERT tokenizer. Again, we used our library's integration with `transformers` to add a column to our DataFrame of tokens containing BERT embeddings at each token position as tensors. Then we used `scikit-learn` [PVG+11] to train an ensemble of 17 token classification models over multiple different Gaussian random projections. By cross-referencing the outputs of these models, again using Pandas and spanner algebra, we were able to identify a large number of additional incorrect labels in the test fold.

We also used Text Extensions for Pandas' integration with Jupyter to build an interface for human review of the suspicious labels that our analysis of model outputs had flagged. Figure 11 shows this interface in action.

The code that we used in this paper is available as a collection of Jupyter notebooks at https://github.com/CODAIT/text-extensions-for-pandas/tree/master/tutorials/corpus. We are currently working to extend the techniques we developed in order to cover a wider variety of token classification corpora and to incorporate several of the techniques used in our paper into the Text Extensions for Pandas library [MRX+21].

Conclusion

This paper has introduced our library, Text Extensions for Pandas. Text Extensions for Pandas provides a collection of extension data types, NLP-specific operations, and NLP library integrations that turn Pandas DataFrames into a universal data structure for managing the machine data that flows through NLP applications.

Text Extensions for Pandas is freely available as both an installable Python package and as source code. We publish packages on the PyPI and Conda-Forge package repositories. Since our library is implemented in pure Python, these packages work on most operating systems.

The source code for Text Extensions for Pandas is available at https://github.com/CODAIT/text-extensions-for-pandas under version 2 of the Apache license. We welcome community contributions to the code as well as feedback from users about bugs and feature requests.

REFERENCES

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition, 2009.

[cd21] Python core developers. Cpython, 2021. URL: https://github.com/python/cpython.

[Com21] Apache Arrow Committers. Apache arrow, 2021. URL: https://arrow.apache.org/.

[Con21] Spyder Project Contributors. Spyder, 2021. URL: https://github.com/spyder-ide/spyder.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/N19-1423, doi:10.18653/v1/N19-1423.

[dt21a] The D-Tale development team. D-tale, 2021. URL: https://github.com/man-group/dtale.

[dt21b] The Qgrid development team. Qgrid, 2021. URL: https://github.com/quantopian/qgrid.

[Exp21] Explosion.io. SpaCy api documentation: Containers, 2021. URL: https://spacy.io/api/doc.

[FKRV15] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2), May 2015. URL: https://doi.org/10.1145/2699442, doi:10.1145/2699442.

[HH17] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017. URL: http://doi.org/10.5334/jors.148, doi:10.5334/jors.148.

[HMvdWea20] Charles R. Harris, K. Jarrod Millman, and Stéfan J. van der Walt et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[HMVLB20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020. URL: https://doi.org/10.5281/zenodo.1212303, doi:10.5281/zenodo.1212303.

[Hug21] Huggingface. Transformers api documentation: Tokenclassificationpipeline, 2021. URL: https://huggingface.co/transformers/main_classes/pipelines.html#tokenclassificationpipeline.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[Inta] International Business Machines Corp. IBM Watson Discovery. URL: https://www.ibm.com/cloud/watson-discovery.

[Intb] International Business Machines Corp. IBM Watson Natural Language Understanding. URL: https://www.ibm.com/cloud/watson-natural-language-understanding.

[KR18] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/D18-2012, doi:10.18653/v1/D18-2012.

[LB02] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*, 2002.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL: http://arxiv.org/abs/1907.11692, arXiv:1907.11692.

[Mai21a] Stanza Maintainers. Source code for basic data structres in stanza. https://github.com/stanfordnlp/stanza/blob/main/stanza/models/common/doc.py, 2021.

[Mai21b] TensorFlow Text Maintainers. TensorFlow text, 2021. URL: https://github.com/tensorflow/text.

[Mai21c] TensorFlow Text Maintainers. TensorFlow text api documentation, 2021. URL: https://www.tensorflow.org/text/api_docs/python/text.

[MRX+21] Karthik Muthuraman, Frederick Reiss, Hong Xu, Bryan Cutler, and Zachary Eichenberger. Data cleaning tools for token classification tasks. In *DaSH-LA*, 2021.

[NdMG+20] Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajic, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis M. Tyers, and Daniel Zeman. Universal dependencies v2: An evergrowing multilingual treebank collection. *CoRR*, abs/2004.10643, 2020. URL: https://arxiv.org/abs/2004.10643, arXiv:2004.10643.

[pdt21a] The pandas development team. Pandas api documentation: Extending pandas, 2021. URL: https://pandas.pydata.org/pandas-docs/stable/development/extending.html.

[pdt21b] The pandas development team. pandas-dev/pandas: Pandas 1.2.4, April 2021. URL: https://doi.org/10.5281/zenodo.4681666, doi:10.5281/zenodo.4681666.

[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.

[QZZ+20] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. Stanza: A python natural language processing toolkit for many human languages. *CoRR*, abs/2003.07082, 2020. URL: https://arxiv.org/abs/2003.07082, arXiv:2003.07082.

[RC21] Frederick Reiss and Bryan Cutler. Market intelligence with pandas and ibm watson. 2021.

[RM95] Lance A. Ramshaw and Mitchell P. Marcus. Text chunking using transformation-based learning. *CoRR*, cmp-lg/9505040, 1995. URL: http://arxiv.org/abs/cmp-lg/9505040.

[RXC+20] Frederick Reiss, Hong Xu, Bryan Cutler, Karthik Muthuraman, and Zachary Eichenberger. Identifying incorrect labels in the CoNLL-2003 corpus. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 215–226, Online, November 2020. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/2020.conll-1.16, doi:10.18653/v1/2020.conll-1.16.

[SHG+15] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.

[TKSDM03] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003. URL: https://www.aclweb.org/anthology/W03-0419.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Hugging-

face's transformers: State-of-the-art natural language processing, 2020. arXiv:1910.03771.

CLAIMED, a visual and scalable component library for Trusted AI

Romeo Kienzler[‡*], Ivan Nesic[§]

Abstract—CLAIMED is a component library for artificial intelligence, machine learning, "extract, transform, load" processes and data science. The goal is to enable low-code/no-code rapid prototyping by providing ready-made components for various business domains, supporting various computer languages, working on various data flow editors and running on diverse execution engines. To demonstrate its utility, we constructed a workflow composed exclusively of CLAIMED components. For this purpose, we made use of a publicly available Computed Tomography (CT) scans dataset [covidata] and created a deep learning model, which is supposed to classify exams as either COVID-19 positive or negative. The pipeline was built with Elyra's Pipeline Visual Editor, with support for local, Airflow and Kubeflow execution.

Index Terms—Kubernetes, Kubeflow, JupyterLab, Elyra, KFServing, TrustedAI, AI Explainability, AI Fairness, AI Adversarial Robustness

Introduction

In our Hospital Research Department we regularly have Citizen Data Scientists (CDS) [citizends] (in our case, mostly medical doctors) working on large corpora of clinical data. Often, monolithic scripts are used for prototyping, lacking quality and reproducibility. Therefore, in cooperation with CDS we've defined the following requirements for a new way of applying data-driven clinical research:

- low-code / no-code environment for rapid prototyping with visual editing and jupyter notebooks
- seamless scaling during development and deployment
- GPU support
- pre-build components for various business domains
- support for the complete python and R tooling including Apache Spark, TensorFlow, PyTorch, pandas and scikit-learn
- seamless extensibility
- reproducibility of work
- data lineage
- collaboration support

We've evaluated the following software tools but we found that these tools, even when used in conjunction, support only a subset of our requirements: Slurm [slurm], Snakemake [snakemake], QSub [qsub], HTCondor [htcondor], Apache Nifi [nifi], NodeRED [nodered], KNIME [knime], Galaxy [galaxy], Reana [reana], WEKA [weka], Rabix [rabix], Nextflow [nextflow], OpenWDL [openwdl], CWL [cwl] or Cromwell [cromwell].

To not reinvent the wheel but rather fill the gap, we have built an extensible component library to be used in low-code / no-code environments called CLAIMED - the visual **C**omponent **L**ibrary for **A**rtificial **I**nteligence (AI), **M**achine Learning (ML), **E**xtract, Transform, Load (ETL) and **D**ata Science. In the following section we elaborate on the implementation details followed by a description of an exemplary pipeline to showcase the capabilities of CLAIMED. We continue to elaborate on different ideas how CLAIMED can be improved in the "Future Work" section, finally followed by the conclusion.

Implementation

Before we address how CLAIMED fulfills the previously defined requirements and how the exemplary workflow has been constructed, we will introduce some terms and technologies.

Technology breakdown

Containerization and Kubernetes: Virtualization opened up a lot of potential for managing the infrastructure, mainly the ability to run different operating systems on the same hardware at the same time. Next step of isolation can be performed for each of the microservices running on the server, but instead of managing access rights and resources on the host operating system, we can containerize these in separate packages with their own environments. Practical effect of this is that we are running each of the microservices as if they have their own dedicated virtual machine, but without the overhead of such endeavour. This is accomplished by running containers on top of the host operating system. An example of the containerization platform is Docker.

With the opportunity to run a vast number of containers, arose the need of their orchestration. The system needs to be constantly monitored and adjusted so that it stays in a desired state. Containers need to be scaled up and down, the communication has to be managed, they have to be scheduled, authentication needs to be managed, there is the need for load balancing etc. There are multiple optins on the market, but Kubernetes is the market leader in this domain. It was donated to Cloud Native Computing Foundation (CNCF) [cncf] by Google, which means a lot of Google's know-how and years of experience went into it. The system can run on public, on-premises or on hybrid clouds. On-premises installation is very important for institutions dealing

* *Corresponding author: romeo.kienzler@ch.ibm.com*
‡ *IBM, Center for Open Source Data and AI Technologies (CODAIT)*
§ *University Hospital of Basel*

with sensitive data. For IBM, Kubernetes is also strategic. IBM acts as a Kubernetes runtime provider in the cloud and - through the acquisition of RedHat - has become the major vendor for on-premises Kubernetes. IBM is now able to deliver software solutions - so called "Cloud Paks" - on top of Kubernetes, making them run everywhere (hybrid cloud). Therefore, IBM joined CNCF [ibmcncf], and moved all Watson Services to Kubernetes. This makes IBM the 3rd largest committer to Kubernetes. Not only for IBM but also for us, Kubernetes enables the hybrid cloud scenario of transparently moving workload across different on-premises, remote and cloud data centers seamlessly.

Deep Learning with TensorFlow: TensorFlow is the second incarnation of the Google Brain project's scalable distributed training and inference system named DistBelief [tf]. It supports myriad of hardware platforms, from mobile phones to GPU/TPU clusters, for both training and inference. It can even train and run models in browser, without the data ever leaving the user's environment. Apart from being a valuable tool in research domain, it is also being used in demanding production environments. On a development side, representing machine learning algorithms in tree-like structures makes it a good expression interface. Lastly, on the performance vs usability side, both graph and eager modes are supported. Eager mode allows for easier debugging since the code is executed in Python control flow, as opposed to the TensorFlow specific graph control flow [tfeager]. The advantages of graph mode is usage in distributed training, performance optimization and production deployment. In-depth analysis of these two modes can be found here [tfbook].

Kubeflow: Having a compute cluster capable of scaling at container level granularity calls for a workflow execution engine leveraging the advantages of containerization and container orchestration by integrating with Kubernetes seamlessly. This is where Kubeflow [kubeflow] kicks in. It is a machine learning pipeline management and execution system running as first class citizen on top of Kubernetes. Beside making use of Kubernetes scalability, it allows for defining reproducible work products as machine learning pipelines, where results and intermediate artifacts of the executions are stored in a metadata repository.

Jupyter Notebooks / JupyterLab: When it comes to collaboration and reproducibility, document centric coding tools like Apache Zeppelin or Jupyter Notebooks are a great choice. Recently, JupyterLab [jupyter] started setting the standard for the research and data science community [jupyter_standard]. Therefore we consider Jupyter Lab not only as an Integrated Development Environment (IDE) but more as a technology standard practitioners love to work with.

Elyra: Visual editing using drag and drop editing in "no code" / "low code" environments is gaining popularity [lowcode]. As a representative of such environments we introduce Elyra. Elyra [elyra] started as a set of extensions for the JupyterLab ecosystem. Here we concentrate on the pipeline editor, developed by IBM in Open Source under supervision of the authors, which allows for expression of machine learning workflows using a drag and drop editor. Inspired by CWL [cwl] and OpenWDL [openwdl], Elyra uses an open and interchangeable, JSON based format to represent the workflows. This allows Elyra to transpile workflows to different execution engines like Kubeflow or Airflow. This means non-programmers can understand and create machine learning workflows on their own without coding and at the same time making use of Kubernetes massive scalability. Elyra also ships with a browser extension for visualizing such pipelines in the browser (e.g. from a github repository) to improve collaboration.

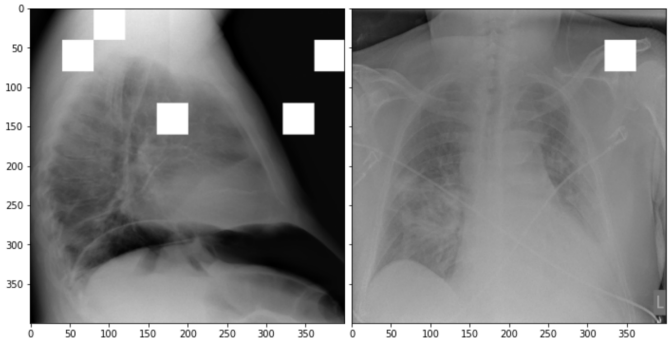

Fig. 1: *Example on how LIME helps to identify classification relevant areas of an image.*

AI Explainability: Despite the good performance, deep learning models are viewed as being black box approaches. Technically, deep learning models are a series of non-linear feature space transformations, but an intuitive understanding of each of the individual processing steps is not trivial. There are techniques with which we can look over a deep learning model's shoulder. The one we are using is called LIME [lime]. LIME takes the existing classification model and permutes images taken from the validation set (therefore the real class label is known to LIME) as long as a misclassification is happening. That way LIME can be used to create heat maps as image overlays to indicate regions of images which are most relevant for the classifier. In other words, we identify regions of the image the classifier is looking at.

As Fig. 1 illustrates, the most relevant areas in an image for classifying for COVID-19 are areas containing bones over lung tissue which indicates a problem with that particular classifier.

AI Fairness and Bias: "Bias is a disproportionate weight in favor of or against an idea or thing, usually in a way that is closed-minded, prejudicial, or unfair" [bias]. But what we want from our model is to be fair and unbiased towards protected attributes like race, age, socioeconomic status, religion and so on. So wouldn't it be easier if we just "hid" those columns from the model during the training? Unfortunately the problem is convoluted. Protected attributes are often encoded inside the other attributes (latent features). For example, race, religion and socioeconomic status are latently encoded in attributes like zip codes, contact methods or types of products purchased. Therefore, fairness assessment and bias detection is quite challenging. Luckily, a huge number of single number metrics exist to assess bias in data and models. Here, we are using the AIF360 [aif360] library. IBM donated it to the Linux Foundation AI, which puts it under open governance.

AI Adversarial Robustness: Another pillar of Trusted AI is adversarial robustness. For example, as researchers found out, adversarial noise can be introduced in data (data poisoning) or models (model poisoning) to influence models decisions in favor of the adversarial. Libraries like the Adversarial Robustness Toolbox ART [art] support all state-of-the-art attacks and defenses.

Requirements and System Architecture

In the following section we cover the system architecture and it's requirements. There are two major components: execution engine and integrated tools.

Execution Engine: An execution engine takes a pipeline description and executes it on top of physical machines, reads

Requirement	KF	AF	Slurm	SM	Qsub	HTCondor	Reana
Kubernetes Support	X	X		X		X	X
GPU support	X	X	X	X	X	X	X
Component Library	X						
Reproducibility	X	X		X		X	X
Data Lineage	X						X

TABLE 1: Fulfilment of requirements for execution engines. (Abbreviations: KF=Kubeflow, AF=Airflow, SM=Snakemake)

Requirement	Nifi	NodeRED	KNIME	Galaxy	Elyra
Kubernetes Support				X	X
GPU support				X	X
Component Library	X	X	X	X	X
Reproducibility	X		X	X	X
Data Lineage	X			X	X
Visual Editing	X	X	X	X	X
Jupyter Notebooks					X

TABLE 2: Fulfilment of requirements for integrated tools.

source data and creates output data. The following requirements have been defined in order to assess the adequacy of the execution engine.

- Kubernetes Support
  We defined Kubernetes as the lowest layer of abstraction because that way the executor layer is agnostic of the underlying Infrastructure as a service (IaaS) architecture. In addition, Kubernetes provides better resource utilization if multiple pipelines are run in parallel on the system. We can consume Kubernetes as a service (aaS) offered by a variety of Cloud providers like IBM, Amazon, Google, Microsoft, OVH or Linode. A lot of workload for this particular project has been envisioned to be outsourced to SciCore [scicore] - a scientific computing data center part of the Swiss Personalized Health Network (SPHN) [sphn] and the Swiss Institute of Bioinformatics [sib]. Best to our knowledge, their cluster runs on OpenStack and provides Kubernetes as part of it (Magnum). University Hospital of Basel has on-premises RedHat OpenShift platform.
- GPU support
  GPU support is essential since a large fraction of the workload is training of deep learning neural networks with TensorFlow and PyTorch. Training those models on CPU doesn't make sense economically and ecologically.
- Component Library
  Predefined, ready to use components, are convenient to use, they save time and, if well tested, reduce the probability of an error. Kubeflow for example has components for parallel training of TensorFlow models (TFJob), parallel execution of Apache Spark jobs as a pipeline step, parallel hyperparameter tuning (Katib) and model serving (KFServing/ KNative)
- Reproducibility
  From a legal point of view, in certain domains, it is necessary to reconstruct a certain decision, model or output dataset for verification and audit. Therefore the ability to reproduce and re-run a pipeline is a critical requirement. Of course, there are other examples where this is imperative, like in science.
- Data Lineage
  Although a subset of reproducibility, Data Lineage is a crucial feature when it comes to visualizing the changes the datasets went through during the pipeline execution.

Integrated tools: Integrated tools are tools which include a visual data flow editor, a component library and an execution engine. Prominent candidates in the open source space are Apache Nifi, NodeRED, KNIME and Galaxy.

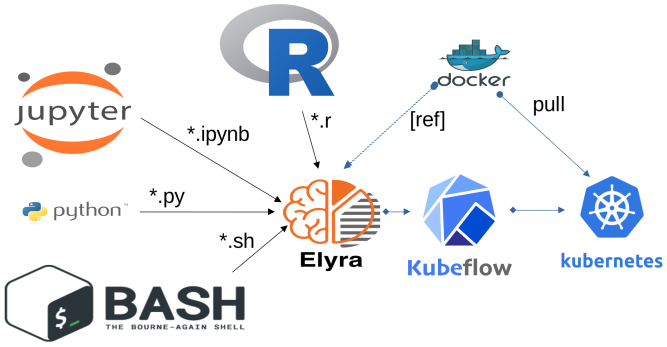

Fig. 2: Runtime architecture of CLAIMED.

The following additional requirements have been defined for a suitable tool:

- Low-Code/No-Code/Visual Editing
  Citizen data scientists (in our demo example, medical doctors) need to work with the tool, so visual editing is necessary. But apart from being a visual editing tool, support for creating custom pipeline components on the fly using R and python is necessary as well.
- Jupyter Notebooks
  Researchers in general like to implement tasks in jupyter notebooks. This makes support for JupyterLab, as well as having an easy way of making Jupyter notebooks part of the data processing pipeline, a key requirement.

Final technology choice: As it can be seen from the tables 1 and 2, only Kubeflow on the execution engine side, and Elyra as the integrated tool are capable of covering all of the requirements. Therefore we select this pair as our primary technology choice.

Elyra's pipeline editor supports drag and drop functionality, for adding arbitrary scripts (shell, R, python) and Jupyter notebooks to the canvas. Each script gets a container image assigned to be executed in. At the moment, Elyra supports pipeline submissions to Airflow and Kubeflow.

Together with Kubeflow and JupyterLab (where Elyra runs as an extension), all our requirements are fulfilled.

As it can be seen on Figure 2, Elyra - specifically the pipeline editor of the Elyra Extension to JupyterLab - allows for visually building data pipelines with a set of assets like notebooks and scripts dragged onto a canvas and transparently published to Kubeflow, as a Kubeflow pipeline.

The only thing missing is a set of re-usable notebooks for different kinds of tasks and this is where CLAIMED comes in.

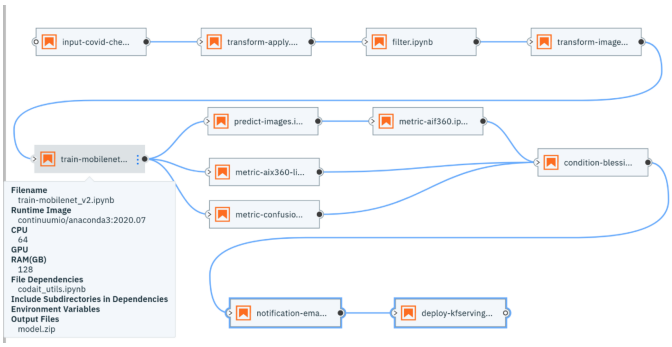

Fig. 3: *The exemplary TrustedAI pipeline for the health care use case.*

We've published CLAIMED as an open source library [complib]. In the next sections we will introduce the demo use case, along with how components found in CLAIMED have been used to implement this pipeline.

System Implementation and Demo Use Case

A TrustedAI image classification pipeline

As mentioned, pipelines are a great way to introduce reproducibility, scaling, auditability and collaboration in machine learning. Pipelines are often a central part of a MLOps strategy. This holds for TrustedAI pipelines too, since reproducibility and auditability are even more important in this case. Figure 3 illustrates the exemplary TrustedAI pipeline we have built using the component library and Figure 4 is a screenshot taken from Kubeflow displaying the pipeline after finishing it's run.

Pipeline Components

This section exemplifies each existing category with at least one component which has been used for this particular pipeline. There are also other components that are not part of the pipeline, so they are not introduced here. Please note that the core feature of our software is threefold:

- the CLAIMED component library
- Elyra with it's capability to use CLAIMED to create a pipeline and push it to Kubeflow
- the pipeline itself

Input Components: There are input components for different types of data source, like files and databases.

In this particular case, we're pulling data directly from the GitHub repository via a public and permanent link [covidata]. We only pull the metadata.csv and images directory.

Transform Components: Sometimes, transformations on the metadata, or any other structured dataset, are necessary. Therefore, we provide a generic transformation component - in the example, we used it to change to format of the categories as the original file contained forward slashes which made it hard to use on the underlying operating system. This is performed by specifying a column name and a function that has to be applied.

Filter Components: Similar to changing content of rows in a dataset, removing rows is also a common task in data engineering. The filter stage allows doing exactly that. It is enough to provide a predicate - specifically for our case the predicate `~metadata.filename.str.contains('.gz')` removes invalid images.

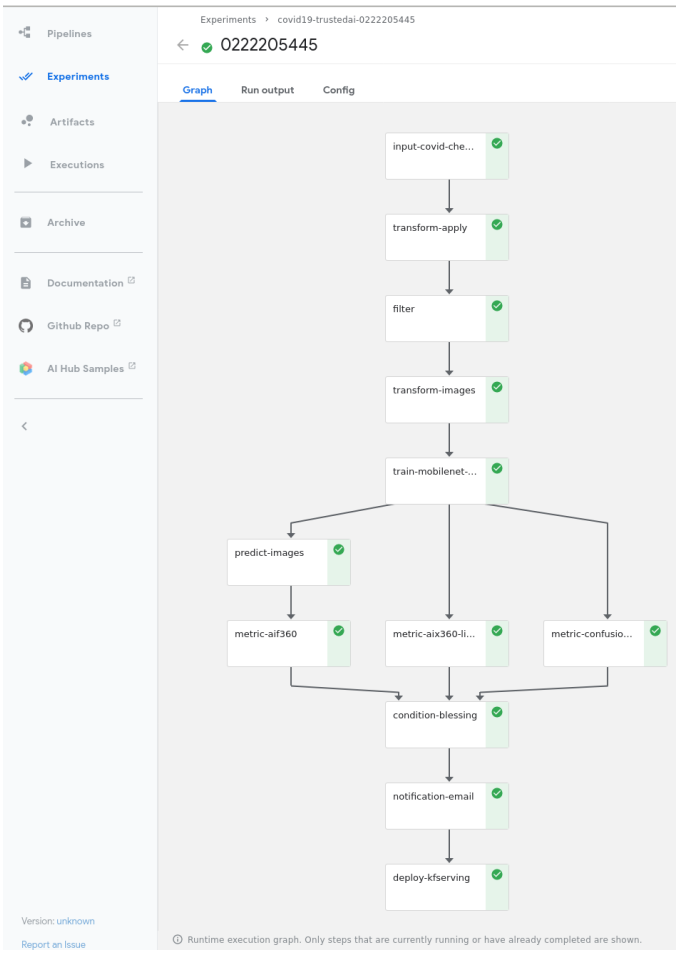

Fig. 4: *The pipeline once executed in Kubeflow.*

```
data
├── No Finding
│   ├── 000001-3.jpg
│   ├── 000001-6.jpg
├── Pneumonia_Viral_COVID-19
│   ├── 00870a9c.jpg
│   ├── 01E392EE-69F9-4E33-BFCE-E5C968654078.jpeg
```

Fig. 5: *Example of directory structure supported by TensorFlow Dataset API.*

Image Transformer Components: One supported standard for the conversion of image datasets into the TensorFlow's dataset supported format, is to organize images into directories representing their classes [tfimgprep]. TensorFlow Dataset is an API that allows for a convenient way to create datasets from various input data, apply transformations and preprocessing steps and make iteration over the data easier and memory efficient [tfdataset].

In our example, the data isn't in the required format. It is organized as a directory full of images and alongside it is a CSV file which defines the attributes. Available attributes are exam finding, sex and age, from which we only require the finding for our example. The images are then arranged by following the previously described directory structure, as illustrated by Fig. 5. After performing this step, the data can be consumed by the Tensorflow Dataset API.

```
exec('input_shape = ('+image_shape+',3)')

model = tf.keras.applications.MobileNetV2(
    input_shape=input_shape, alpha=1.0, include_top=False,
    input_tensor=None, pooling=None, classes=num_classes,
    classifier_activation='softmax'
)
model = my_net(model, num_classes=num_classes)
```

Fig. 6: *Source code of the wrapped training component.*

Training Components: Understanding, defining and training deep learning models is not a simple task. Training a deep learning image classification model requires a properly designed neural network architecture. Luckily, the community trends towards predefined model architectures, which are parameterized through hyper-parameters. At this stage, we are using the MobileNetV2, a small deep learning neural network architecture with the set of the most common parameters. It ships with the TensorFlow distribution - ready to use, without any further definition of neurons or layers. As shown in Figure 6, only a couple of parameters need to be specified.

Although possible, hyper-parameter search is not considered in this processing stage. The reason being, we want to make use of Kubeflow's hyper-parameter search capabilities leveraged through Katib [katib] in the future.

Evaluation Components: A model needs to be evaluated before it goes into production. Evaluating classification performance against the target labels has been a common metric since the early days of machine learning, therefore we have also developed evaluation components, with confusion matrix support for instance. But taking TrustedAI measures into account is a newly emerging practice. Therefore, components for AI Fairness [aif360], AI Explainability [aix360] and AI Adversarial Robustness [art] have been also added to the component library.

Blessing Components: In Trusted AI (but not limited to) it is important to obtain a blessing of assets like generated data, models or reports to be published and used by other subsystems or humans. Therefore, a blessing component uses the results of the evaluation components to decide if the assets are ready for publishing.

Publishing Components: Depending on the asset type, publishing means either persisting a dataset to a data store, deploying a machine learning model for consumption of other subsystems, or publishing a report to be consumed by humans. Here, we exemplify this category by a KFServing [kfserving] component which publishes the trained TensorFlow deep learning model to Kubernetes. KFServing, on top of KNative, is particularly interesting as it draws from Kubernetes capabilities, like canary deployment and scalability (including scale to zero), in addition to built-in Trusted AI functionality.

Future Work

We have financial support to add functionality to CLAIMED in multiple dimensions. Below we give a summary of the next steps.

Extend component library

To this date, at least one representative component for each category has been released. Components are added to the library on a regular basis. The components due to be published are: Parallel Tensorflow Training with TFJob, Parallel Hyperparameter Tuning with Katib and Parallel Data Processing with Apache Spark.

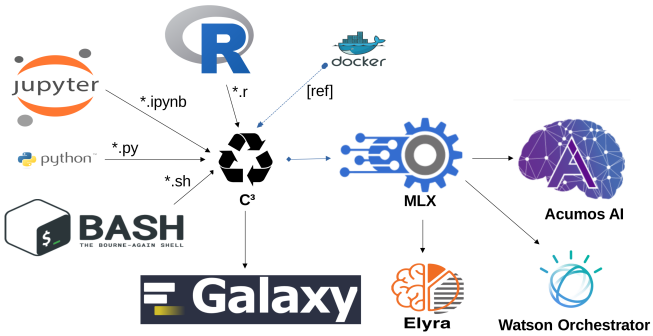

Fig. 7: *C3 - The CLAIMED Component Compiler transpiles and publishes pipeline components for different target platforms*

Component exporter for Kubeflow

Containerizing notebooks and scripts is a frequent task in the data science community. In our environment, this involves attaching the arbitrary assets, like jupyter notebooks and scripts, to a container image and then transpiling a Kubeflow component out of it. We are currently in the process of implementing a tool that would facilitate this workflow. The name of the tool is C3 [c3], and it stands for CLAIMED component compiler. Currently, transpiling from notebooks to Kubeflow Pipeline components is supported. In addition, publishing these components to component repositories will also be possible. C3 already supports publishing components to Machine Learning Exchange (MLX) [mlx], an open source asset repository for notebooks, pipelines, data sets, machine learning models and pipeline components. Figure 7 illustrates the concept.

Import/Export of components to/from Galaxy

As seen in Table 2, Galaxy covers a majority of our requirements already. Unfortunately, Galaxy components - called "tools" - are very skewed towards genomics. Adding new components and extending functionality onto other domains would make the tool interesting for a wider audience. Reverse is also true, the existing component library Galaxy is extensive, well established and tested. It makes sense to automatically transpile those tools as components into CLAIMED. We are currently looking into adding import/export support between CLAIMED and Galaxy into C3.

UX improvements of the Elyra pipeline editor

The components are isolated, so only explicitly shared information can be put into context for all of them. In order for the components' executor, e.g. Kubflow, to do this, it must be provided a configuration. We envision for Elyra to automatically deduce interesting parameters from the code and from the environment, upon which it would create dynamic forms. For example, fields like checkboxes and dropdowns where one can select input and output files mentioned in the code. Currently, only environment variables are provided in a rudimentary UI with one text field per variable. One proposal is to introduce an optional configuration block to the scripts and notebooks. It would then be interpreted by Elyra and the appropriate UI would be rendered.

One successful example of such implementation is Galaxy's UI [galaxy_ui]. A complex UI behavior is expressed by XML configuration. So we are also exploring an option of either using Galaxy's XML Schema or defining a new one and support the transformation from one into the other.

Add CWL support to the Elyra pipeline editor

CWL is a powerful workflow expression language supported already by various tools we've evaluated. Currently, Elyra uses its own, proprietary pipeline representation format. Adding support of CWL to Elyra would improve interoperability between different software components. For example, the Reana execution engine used in the particle physics community, and Galaxy (partially) already support CWL. This means it would be possible to export pipelines from Elyra to Reana, without the need of transpiling the pipeline. Alternatively, Elyra could integrate export and import of CWL into its pipeline editor.

Import 3rd party component libraries

Since the only thing needed for arbitrary code to become a CLAIMED component is to be wrapped in a container image and to be assigned with meta data, it is possible for 3rd party component libraries like those from KNIME or Nifi and to be imported into CLAIMED. This also holds true for Kubeflow components. It is also possible to wrap different components from KNIME, Nifi or similar tools in this manner and use it within Elyra, as well as in the other execution engines CLAIMED supports.

Create more (exemplary) pipelines

At the moment, CLAIMED ships with three exemplary pipelines. The health care inspired TrustedAI pipeline which was covered in this paper, a pipeline to visualize and predict soil temperature from a historic data set and an IoT sensor data analysis pipeline. The next pipeline in line is a genomics pipeline for the Swiss Institute of Bioinformatics affiliates University Hospital Berne/Berne University and potentially for particle physics at CERN.

Conclusion

We've build a trustable, low-code, scalable and open source component library, targeting visual data pipeline systems. We've showcased the library's capabilities by building a domain specific pipeline on Elyra, an emerging visual pipeline editor and running it on widely used Kubeflow execution engine. We believe that future import/export functionality of CLAIMED will improve reproducibility of data centric work even further.

References

[art] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Beat Buesser, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian M. Molloy, Ben Edwards *Adversarial Robustness Toolbox*, arXiv:1807.01069, November 2019

[aif360] Rachel K. E. Bellamy et al. *AI Fairness 360: An Extensible Toolkit for Detecting, Understanding, and Mitigating Unwanted Algorithmic Bias*, arXiv:1810.01943, October 2018

[aix360] Vijay Arya et al. One Explanation Does Not Fit All: A Toolkit and Taxonomy of AI Explainability Techniques, arXiv:1909.03012, September 2019

[bias] Steinbock, Bonnie (1978). *Speciesism and the Idea of Equality*, Philosophy, 53 (204): 247–256, doi:10.1017/S0031819100016582

[cncf] Cloud Native Computing Foundation, https://www.cncf.io. Last accessed 18 Feb 2021

[complib] https://github.com/elyra-ai/component-library

[elyra] Elyra AI, https://github.com/elyra-ai. Last accessed 18 Feb 2021

[kubernetes] David Bernstein et al. *Containers and Cloud: From LXC to Docker to Kubernetes*, IEEE Cloud Computing (Volume: 1, Issue: 3), September 2014)

[jupyter] Thomas Kluyver et al. *Jupyter Notebooks – a publishing format for reproducible computational workflows*, Positioning and Power in Academic Publishing: Players, Agents and Agendas, 87-90, doi:10.3233/978-1-61499-649-1-87, 2016

[kfserving] Clive Cox and Dan Sun and Ellis Tarn and Animesh Singh and Rakesh Kelkar and David Goodwin, *Serverless inferencing on Kubernetes*, Workshop on "Challenges in Deploying and Monitoring Machine Learning System" at ICML 2020

[lime] Marco Tulio Ribeiro et al. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, pp. 1135–1144 (2016), doi:10.1145/2939672.2939778

[kubeflow] Debo Dutta and Xinyuan Huang, *Consistent Multi-Cloud AI Lifecycle Management with Kubeflow*, OpML, 2019

[katib] George et al. *A Scalable and Cloud-Native Hyperparameter Tuning System*, arXiv:2006.02085, June 2020

[tf] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, arXiv:1603.04467v2, March 2016

[ibmcncf] IBM joining CNCF, https://developer.ibm.com/technologies/containers/blogs/ibms-dedication-to-open-source-and-its-involvement-with-the-cncf Last accessed 18 Feb 2021

[ect] https://github.com/cloud-annotations/elyra-classification-training/tree/developer_article

[slurm] "Yoo, Andy B. and Jette, Morris A. and Grondona, Mark, *SLURM: Simple Linux Utility for Resource Management*, Job Scheduling Strategies for Parallel Processing, Springer, 2003

[snakemake] Köster, Johannes and Rahmann, Sven, *Snakemake—a scalable bioinformatics workflow engine*, Journal of Bioinformatics, Number 19, Volume 28, Pages 2520-2522}, August 2012

[qsub] https://en.wikipedia.org/wiki/Qsub

[htcondor] E M Fajardo et al, *How much higher can HTCondor fly?*, 2015 Journal of Physics.: Conference Series, Volume 664, June 2014

[galaxy] Enis Afgan et al. *The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update*, Nucleic Acids Research, 46):W537-W544, doi:10.1093/nar/gky379, July 2018

[reana] Tibor Šimko et al. *REANA: A System for Reusable Research Data Analyses*, 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018), (214):06034, doi:10.1051/epjconf/201921406034, September 2019

[nifi] https://nifi.apache.org/

[nodered] Z. Chaczko and R. Braun, *Learning data engineering: Creating IoT apps using the node-RED and the RPI technologies*, 16th International Conference on Information Technology Based Higher Education and Training (ITHET), 1-8, doi: 10.1109/ITHET.2017.8067827, 2017

[knime] Michael R. Berthold et al. *KNIME - the Konstanz information miner: version 2.0 and beyond*, ACM SIGKDD Explorations Newsletter, (11):26–31, doi:10.1145/1656274.1656280, June 2009

[weka] Mark Hall et al. *The WEKA data mining software: an update*, ACM SIGKDD Explorations Newsletter, (11):10–18, doi:10.1145/1656274.1656278, June 2009

[rabix] Gaurav Kaushik et al., *RABIX: AN OPEN-SOURCE WORKFLOW EXECUTOR SUPPORTING RECOMPUTABILITY AND INTEROPERABILITY OF WORKFLOW DESCRIPTIONS*, Proceedings of the Pacific Symposium on Biocomputing 2017, (22):154-165, doi:10.1142/9789813207813_0016, November 2017

[nextflow] Di Tommaso, P., Chatzou, M., Floden, E. et al. *Nextflow enables reproducible computational workflows.*, Nature Biotechnology, (35):316–319, doi:10.1038/nbt.3820, 2017

[openwdl] https://openwdl.org/

[cwl] https://www.commonwl.org/

[cromwell] https://cromwell.readthedocs.io/en/stable/

[covidata] Joseph Paul Cohen et al. *COVID-19 Image Data Collection: Prospective Predictions Are the Future*, arXiv:2006.11988, 2020

[tfeager] Akshay Agrawal et al. *TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning*, Proceedings of the 2nd SysML Conference, arXiv:1903.01855, 2019

[tfdataset] Steven W. D. Chien et al. *Characterizing Deep-Learning I/O Workloads in TensorFlow*, IEEE/ACM 3rd International Workshop on Parallel Data Storage - Data Intensive Scalable Computing Systems (PDSW-DISCS), doi:10.1109/PDSW-DISCS.2018.00011 2018

[tfimgprep] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory

[galaxy_ui] https://github.com/bgruening/galaxytools/blob/c1027a3f78bca2fd8a53f076ef718ea5adbf4a8a/tools/sklearn/pca.xml#L75

[c3] https://github.com/romeokienzler/c3

[tfbook] Romeo Kienzler and Jerome Nilmeier, *What's New In TensorFlow 2.x?*, O'Reilly Media, Inc., ISBN: 9781492073710, July 2020

[vscode] https://code.visualstudio.com/

[scicore] https://scicore.unibas.ch/

[sphn] https://sphn.ch/

[sib] https://www.sib.swiss/

[citizends] Mullarkey, Matthew T. et al., *Citizen Data Scientist: A Design Science Research Method for the Conduct of Data Science Projects*, Extending the Boundaries of Design Science Theory and Practice, 191-205, Springer International Publishing, ISBN 978-3-030-19504-5, 2019

[jupyter_standard] Perkel, Jeffrey M. *Why Jupyter is data scientists' computational notebook of choice.* Nature, Volume 563, Number 7732, Page 145+, 2018

[lowcode] Apurvanand Sahay et al. *Supporting the understanding and comparison of low-code development platforms*, 171-178, IEEE 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), doi:10.1109/SEAA51224.2020.00036, 2020

[mlx] https://github.com/machine-learning-exchange/mlx, Accessed: 29 of June, 2021

PyCID: A Python Library for Causal Influence Diagrams

James Fox[‡*], Tom Everitt[§], Ryan Carey[‡], Eric Langlois[¶], Alessandro Abate[‡], Michael Wooldridge[‡]

Abstract—Why did a decision maker select a certain decision? What behaviour does a certain objective incentivise? How can we improve this behaviour and ensure that a decision-maker chooses decisions with safer or fairer consequences? This paper introduces the Python package *PyCID*, built upon *pgmpy*, that implements (causal) influence diagrams, a widely used graphical modelling framework for decision-making problems. By providing a range of methods to solve and analyse (causal) influence diagrams, *PyCID* helps answer questions about behaviour and incentives in both single-agent and multi-agent settings.

Index Terms—Influence Diagrams, Causal Models, Probabilistic Graphical Models, Game Theory, Decision Theory

Introduction

Influence-diagrams (IDs) are used to represent and analyse decision making situations under uncertainty [HM05], [MIMH+76]. Like Bayesian Networks, IDs have at their core a directed acyclic graph (DAG), but IDs also specify decision and utility nodes. Relationships between variables are given by conditional probability distributions. When these are specified, we call it an influence model (IM). In an IM, a decision-maker selects a distribution over its available actions at a decision (a decision rule) based on what it knows (the values of its parents in the ID) to maximise its expected utility. To demonstrate, consider the following example:

Grade Prediction: To decide who to admit, a university uses a model to predict the grades of applicants based on information in their application forms.

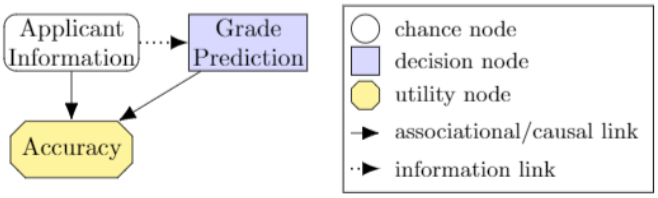

Fig. 1: *A (C)ID for the* ***Grade Prediction*** *example.*

Figure 1 shows the DAG for this example, which displays clearly the structure of the decision situation. The decision being made by an agent, the model, is the grade prediction (decision node). The agent selects a decision rule for this decision, based on information about the applicant (chance node), in order to optimise their prediction accuracy (utility node). The edges denote associational relationships in the case of a statistical IM, but denote causal links in causal influence models (CIMs). This difference in semantics [ECL+21] allows one to use CIMs to query the effect of causal interventions and provides a setting to ask counterfactual questions [Pea09]. (C)IMs have also been extended to multi-agent settings by [KM03], [HFE+21], and [HFE+].

Statistical and causal IDs have shown promise for a wide variety of applications. In business and medical decision making, statistical IDs provide a simple yet powerful model for optimising decisions by making assumptions explicit and revealing what information is relevant [Góm04], [KM08]. Moreover, for the design of safe and fair AI systems, causal IDs have been used to help predict the behaviour of agents arising due to their incentives in an environment [ECL+21], [CLEL20], [EHKK21], [Hol20], [EKKL19], [LE21], and [CVH20]. Nevertheless, although Python libraries exist for Bayesian networks, perhaps most prominently *pgmpy* [AP15], these libraries lack specific support for IDs. We found two Python wrappers of C++ influence diagram libraries: *pyAgrum* [DBDSMW20] and *PySMILE* [Bay]. These were limited by usability (hard to install), maintainability (using multiple languages) and versatility (they did not cover multi-agent or causal IDs). A Python library that focuses on implementing statistical and causal IDs is therefore needed to ensure their potential application can be explored, probed, and fully realised.

Consequently, this paper introduces *PyCID*[1], a Python library built upon *pgmpy* [AP15] and *NetworkX* [HSS08], which implements IDs and IMs (including their causal and multi-agent variants) and provides researchers and practitioners with convenient methods for analysing decision-making situations. *PyCID* can solve single-agent (C)IMs, find Nash equilibria in multi-agent (C)IMs, and compute the effect of causal interventions in CIMs (e.g., fixing the prediction model in Figure 1 to always predict a high grade regardless of the applicant's information). *PyCID* can also find which variables in an ID admit incentives. For example, positive value of information [How66] and value of control [Sha86] tell us when an agent can benefit from observing or controlling a variable. Meanwhile, other incentives concepts, recently proposed in [ECL+21], reveal which variables it can be instrumentally useful to control and when a decision-maker benefits from responding to a variable. Reasoning patterns are a related concept in multi-agent IDs: they analyze why a decision-maker would care about a decision [PG07], and these can also be

* *Corresponding author: james.fox@cs.ox.ac.uk*
‡ *University of Oxford*
§ *DeepMind*
¶ *University of Toronto*

computed in *PyCID*.

The first two sections of this paper provide the necessary background on (C)IDs and describe the architecture of the *PyCID* library. We then move to showcasing some of *PyCID*'s features through applications for discovering agent incentives and analysing games. In the * Instantiating Causal Influence Diagrams* section, we demonstrate how to instantiate a (C)ID for the **Grade Prediction** example in *PyCID*. In the *Analysing Incentives* section, we demonstrate how to find the nodes which admit value of information, response, value of control, or instrumental control incentives for more complex (C)IDs. We then turn to multi-agent (C)IDs (MA(C)IDs) and show how to use *PyCID* to compute Nash equilibria. Next, we explain how *PyCID* can construct random (MA)CIDs. Finally, we discuss the future of *PyCID*.

Background

Notation

Throughout this paper, we will use capital letters, X, for random variables and let $dom(X)$ denote their domain. An assignment $x \in dom(X)$ to X is an instantiation of X denoted by $X = x$. $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of variables with instantiation $\mathbf{x} = \{x_1, \ldots, x_n\}$. We also let $\mathbf{Pa}_V$ denote the parents of a node V in a (MA)CID and $\mathbf{pa}_V$ be the instantiation of $\mathbf{Pa}_V$. Moreover, we define $\mathbf{Desc}_V$ and $\mathbf{Fa}_V := \mathbf{Pa}_V \cup \{V\}$ to be the descendants and family of V. We use subscripts to index the elements of a set and, in a multi-agent setting, superscripts to indicate a player $i \in \mathbf{N}$; e.g., the set of decisions belonging to player i is $\mathbf{D}^i = \{D^i_1, \ldots, D^i_n\}$.

Causal Influence Diagrams

A *Bayesian network* is a model consisting of a directed acyclic graph (DAG) and a joint distribution that is Markov compatible with that graph [Pea09]. The nodes in the DAG denote random variables and the directed edges represent the associational relationships between them. To parameterise the DAG and encode the joint distribution, each random variable, V, in the DAG is assigned a conditional probability distribution (CPD), $P(V|\mathbf{Pa}_V)$, dependent on its set of graphical parents, $\mathbf{Pa}_V$. Taken together, these CPDs define the Bayesian network's joint distribution.

A *causal Bayesian network* is a Bayesian network where the directed edges in the DAG now represent every causal relationship between the Bayesian network's variables. This enables the model the ability to answer questions about the effect of causal interventions from outside of the system.

Causal Influence Diagrams (CIDs) are DAGs where the nodes are partitioned into chance, decision, and utility nodes and the edges adopt the same causal semantics as causal Bayesian networks [ECL+21]. Causal Influence models (CIMs) are parameterised CIDs where, at the outset, the CPDs for chance and utility nodes are defined, but only the domains for the decision variables are fixed.

Definition 1 [ECL+21] A **Causal influence Diagram (CID)** is a directed acyclic graph $(\mathbf{V}, \mathbf{E})$ where the set of vertices $(\mathbf{V})$ connected by directed edges $(\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V})$ are partitioned into chance $(\mathbf{X})$, decision $(\mathbf{D})$, and utility $(\mathbf{U})$ nodes. Utility nodes lack children.

Definition 2 [ECL+21] A **Causal influence Model (CIM)** is a tuple $(\mathbf{V}, \mathbf{E}, \theta)$ where $(\mathbf{V}, \mathbf{E})$ is a CID and $\theta \in \Theta$ is a particular parametrisation over the nodes in the graph specifying for each node $V \in \mathbf{V}$ a finite domain $dom(V)$, for each utility node $U \in \mathbf{U}$ a real-valued domain $dom(U) \subseteq \mathbb{R}$, and for every chance and utility node a conditional probability distribution (CPD) $P(V \mid \mathbf{Pa}_V)$.

Multi-agent Causal Influence Diagrams (MACIDs) partition decision and utility nodes further into sets associated with each agent. In a (MA)CID, a decision rule, $\pi_D(D|\mathbf{Pa}_D)$, is a probability distribution over the actions available at decision node D conditional on the value of its parents in the graph, $\mathbf{Pa}_D$. A policy, π^i, assigns decision rules to all of agent i's decision nodes, and, in a MACIM, a policy profile, π, assigns policies to every agent. In a (MA)CID, each agent i's expected utility, $\mathscr{U}^i_{\mathscr{M}}(\pi)$, under a policy (profile) π is the sum of the expected values of their utility nodes.

Package Architecture

In this section, we outline the structure (Figure 2) and describe the key classes of the *PyCID* library[2].

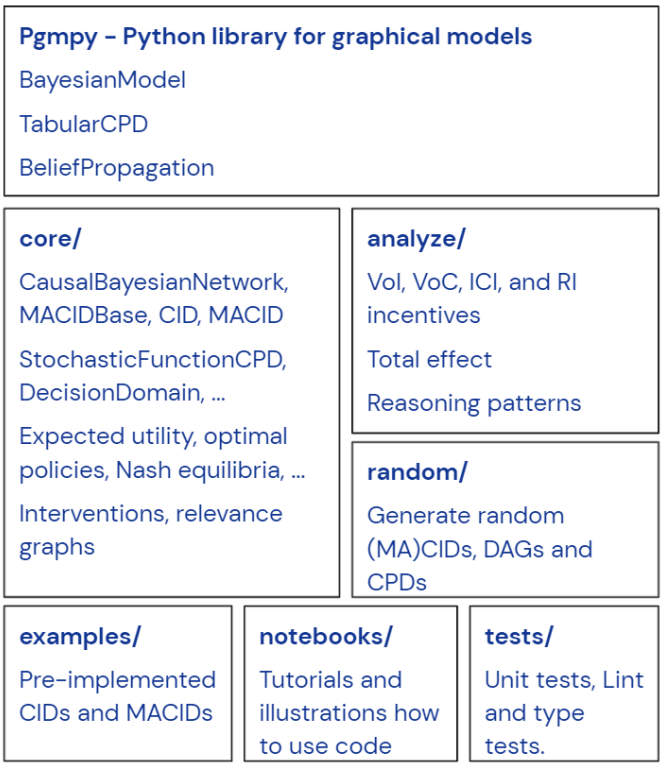

Fig. 2: *An overview of* PyCID*'s file structure.*

Installation

PyCID is released under the *Apache License 2.0*. It requires *Python 3.7* or above, but only depends on *Matplotlib* [Hun07], *NetworkX* [HSS08], *NumPy* [HMvdW+20], and *pgmpy* [AP15]. It can be downloaded and installed in a Python virtual environment or in a Conda environment using:

```
python3 -m pip install pycid
```

PyCID is under continual development and so one can install the latest developmental package using a git checkout from the PyCID repository on GitHub: https://github.com/causalincentives/pycid.

1. This paper describes *PyCID* version 0.2.6.

2. *PyCID* is under continued development, so more features will be added over time. Any updated documentation may be found in the repository's README file.

Classes Inherited from pgmpy

PyCID's key classes inherit from *pgmpy*'s `BayesianModel`, `TabularCPD`, and `BeliefPropagation` classes [AP15]. The `BayesianModel` class represents a *Bayesian network* and CPDs are assigned to each random variable in the model using instances of the `TabularCPD` class. These CPDs define the *Bayesian Network*'s joint distribution and the `BeliefPropagation` class is then used to perform probabilistic inference on a `BayesianModel` object; for instance, one can query the probability that node V takes value v given some instantiation of other variables in the DAG (known as a *context*).

The pycid.core module

PyCID's base class is `CausalBayesianNetwork`. This class inherits from *pgmpy*'s `BayesianModel` and represents a *causal Bayesian network*. In particular, it extends `BayesianModel` by adding the ability to query the effect of *causal interventions*. It also adds methods for determining the expected value of a variable for a given *context* (again under an optional *causal intervention*) and for plotting the DAG of the *Causal Bayesian Network* using *NetworkX* [HSS08]. CPDs for a `CausalBayesianNetwork` object can be defined using *pgmpy*'s `TabularCPD` class, but we also allow relationships to be specified more directly with stochastic functions (under the hood, these are implemented via a `StochasticFunctionCPD` class). This can be used to specify relationships between variables with a stochastic function, rather than just with a probability matrix (see the **Instantiating Causal Influence Diagrams** section). `CausalBayesianNetwork` also has an inner class, `Model`, which keeps track of CPDs and domains for all `CausalBayesianNetwork` objects' variables in the form of a dictionary.

The `MACIDBase` class, which inherits from `CausalBayesianNetwork`, provides the underlying methods necessary for single-agent and multi-agent causal influence diagrams. The class includes methods for determining the expected utility of an agent, for finding optimal decision rules and policies, and for finding various new graphical criteria defined in influence diagrams (e.g. r-relevance).

`CID` and `MACID` are classes, inheriting from `MACIDBase`, that represent single-agent and multi-agent (C)IDs and are the models of most concern in *PyCID*. They include methods for finding the optimal policy for an agent in a (C)IM and for finding Nash equilibria [N+50] and subgame perfect Nash equilibria [Sel65] in a MA(C)IM. It is important to highlight here that statistical (i.e., non-causal) single-agent and multi-agent influence diagrams can also be defined as `CID` and `MACID` objects using *PyCID*. In their case, all class methods are permitted except those that involve causal interventions.

The *pycid.core* module also contains functions that exploit relationships between the (MA)(C)ID's variables such as finding all (active) (directed) paths between variables and classes that find the relevance graphs [KM03] associated with `MACIDBase` objects.

PyCID's other modules

The *pycid.analyse* module includes functions for determining incentives in (C)IDs [ECL+21], reasoning patterns in MA(C)IDs [PG07], and a function for computing the *total effect* of intervening on a variable with different values. *pycid.examples* contains pre-implemented (C)IDs and MA(C)IDs, whilst *pycid.random* contains functions for generating random (C)IDs and MA(C)IDs. *pycid.notebooks* contains *jupyter notebooks* with demonstrations of how to use the codebase; these can also be run directly as *Colab notebooks*. Finally, *pycid.tests* houses unit tests for all functions and public class methods.

Instantiating Causal Influence Diagrams

Having covered *PyCID*'s basic library structure, the remaining sections will demonstrate some use cases. We begin, in this section, by instantiating the structure of the simple (C)ID given in the introduction (Figure 1). For many purposes, including finding incentives, the graph is enough for analysis.

A (C)ID for the **Grade Prediction** example is created as an instance of our `CID` class. Its initializer takes a list of edges as its first argument and then two more lists specifying the (C)ID's decision and utility nodes. All other nodes introduced in the edge pairs, which are not decision or utility nodes, are chance nodes. For conciseness, we abbreviate and use P to denote the prediction model's decision node, A for the applicant's information, and Ac to denote the accuracy of the predictions:

```
import pycid
cid = pycid.CID(
    [("A", "P"), ("A", "Ac"), ("P", "Ac")],
    decisions=["P"],
    utilities=["Ac"],
)
cid.draw()
```

The `CID` class method, *draw*, plots this (C)ID (Figure 3) with a node colour and shape convention that matches what is given in Figure 1's legend.

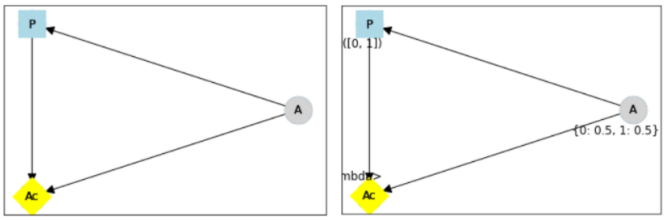

Fig. 3: *A simple (C)ID (Left) and corresponding CIM (Right) plotted using* PyCID.

To then parameterise this (C)ID as a (C)IM by adding a domain for P and CPDs for A and Ac, we pass keyword arguments to the *add_cpds* method:

```
1    cid.add_cpds(
2        A=pycid.discrete_uniform([0, 1]),
3        P=[0, 1],
4        Ac=lambda a, p: int(a == p),
5    )
```

CPDs in *PyCID* can be instantiated directly as `TabularCPD` objects, but more often *PyCID*'s `StochasticFunctionCPD` subclass is used. This provides multiple ways to easily specify how a chance or utility variable's CPD depends on its parents or follows some distribution; it then converts that expression into a `TabularCPD` object under the hood. On line 2 above, we assign variable A a discrete uniform distribution over its domain, $dom(A) = \{0,1\}$; on line 3, we specify $dom(P) = \{0,1\}$; and on the final line, we specify how the value of Ac depends on the values of its parents, A and P. Within the lambda function, other variables are referred to by their lower case form to denote that variable's instantiation. Using a `CID` class method, *solve*, we can

now solve this (C)IM by finding the agent's optimal decision rule for P. This returns the following output, saying that the optimal decision rule for P is to choose action 0 (low grade prediction) when the value of A is 0 (the quality of the application is poor), and action 1 (high grade prediction) when the value of A is 1 (the quality of the application is high):

```
{'P': StochasticFunctionCPD<D>
   {'a': 0}  ->  0
   {'a': 1}  ->  1}
```

If the agent behaves according to this optimal decision rule, we find that their expected utility is 1 using the code below; *expected_utility* accepts optional dictionaries for specifying contexts and causal interventions:

```
solution = cid.solve()
optimal_d_cpd = solution['P']
cid.add_cpds(optimal_d_cpd)
cid.expected_utility(context={}, intervention={})
```

There are several other ways to specify CPDs for variables. For example, on line 1 below, the CPD for A is updated to now follow a Bernoulli(0.8) distribution and line 2 specifies that now Ac just copies the value of P with probability 0.7:

```
1   cid.add_cpds(A=pycid.bernoulli(0.8))
2   cid.add_cpds(Ac=lambda a, p: pycid.noisy_copy(p,
3                  probability=0.7, domain=[0, 1]))
```

Analysing Incentives

In this section, we demonstrate how to use *PyCID* to find which nodes in a single-decision CID admit different types of incentives using their graphical criterion [ECL+21]. In general, a graphical criterion tells you what properties influence models can have based on the influence diagram (i.e, the graph) alone. A graphical criterion takes a graph and several nodes as arguments and returns whether or not the property (in this case the incentive) can occur for those nodes. Incentives are helpful for applications in safety and fairness ([ECL+21], [Hol20]), understanding the behaviour of RL algorithms ([LE21], [EHKK21]), and comparing the promise of different AGI safety frameworks [EKKL19]. We believe that *PyCID* can further mature these enquiries.

PyCID currently finds the following incentives in single-decision CIDs using their graphical criteria:

- Value of Information (VoI)
- Response Incentives (RI)
- Value of Control (VoC)[3]
- Instrumental Control Incentives (ICI)

Value of Information (VoI)

Intuitively, a variable has positive value of information (VoI) if a decision-maker would benefit (get more utility) from observing its value before making a decision:

VoI Definition: For a CIM[4] $\mathscr{M}$, and a node $X \in \mathbf{V} \setminus \mathbf{Desc}_D$, let $\mathscr{M}_{X \not\to D}$ and $\mathscr{M}_{X \to D}$ be $\mathscr{M}$ modified by respectively removing and adding the edge $X \to D$. The **value of information** for X is then $\max_\pi \mathscr{U}^i_{\mathscr{M}_{X\to D}}(\pi) - \max_\pi \mathscr{U}^i_{\mathscr{M}_{X\not\to D}}(\pi)$.

VoI has been applied to a wide array of problems in economics and computer science [BP16]. Although *PyCID*'s function *quantitatative_voi* returns the quantitative VoI of a variable in a CIM, for the remainder of this section we shall focus on its graphical criterion, which depends upon which nodes are **requisite** observations in the CID.

Requisite Observation Graphical Criterion: Let $\mathbf{U}_D \in \mathbf{U} \cap \mathbf{Desc}_D$ be the utility nodes downstream of D. An observation $X \in \mathbf{Pa}_D$ in a single-decision CID is **requisite** if $X \not\perp_{\mathscr{G}} \mathbf{U}_D | (\mathbf{Pa}_D \cup \{D\} \setminus \{X\})$[5].

VoI Graphical Criterion: A single decision CID, $\mathscr{G}$, admits **VoI** for $X \in \mathbf{V} \setminus \mathbf{Desc}_D$ if and only if X is a requisite observation in $\mathscr{G}_{X\to D}$, the graph obtained by adding $X \to D$ to $\mathscr{G}$.

To demonstrate how to find nodes that admit VoI using *PyCID*, we extend the **Grade Prediction** example given in the introduction:

Extended Grade Prediction: [ECL+21] The university wants to admit the brightest students using their grade prediction model, but doesn't want to treat students differently based on their gender (Ge) or race (R). The model uses the gender of the student and the high school (HS) they attended to make its grade prediction. We make the following assumptions:

- Performance at university is evaluated by a student's grades (Gr) and this depends on the quality of education (E) the student received before university (which depends on the high school they attended).
- A student's high school is assumed to be impacted by their race, but not by their gender.

We want to know whether the predictor is incentivised to behave in a discriminatory manner with respect to the students' gender or race. A CID for this example is defined below:

```
cid = pycid.CID(
    [
        ("R", "HS"),
        ("HS", "E"),
        ("HS", "P"),
        ("E", "Gr"),
        ("Gr", "Ac"),
        ("Ge", "P"),
        ("P", "Ac"),
    ],
    decisions=["P"],
    utilities=["Ac"],
)
```

PyCID finds that HS, E, and Gr can all have positive VoI for the predictor model (line 1). We can also display this visually (Figure 4) by passing, as an argument, a lambda function into `CID`'s *draw_property* method (line 2):

```
1   pycid.admits_voi_list(cid, 'P')
2   cid.draw_property(lambda node:
3         pycid.admits_voi(cid, 'P', node))
```

Our implementation of this example in *PyCID* has revealed that there exists a parameterisation of this setup (i.e., a CIM with the given CID) where the model would benefit from knowing the value of one or more of 'High School', 'Education', or the student's true 'Grade' before making a grade prediction.

Response Incentives (RI)

Response incentives (RI) are a related type of incentive and we explain how implementing them in *PyCID* can help improve the

3. Nodes can be specified further as admitting indirect or direct Value of Control.

4. This definition is also valid in (non-causal) statistical influence models.

5. $X \not\perp_{\mathscr{G}} Y \mid \mathbf{W}$ denotes that X is d-connected to Y conditional on the set of nodes in $\mathbf{W}$ and $X \perp_{\mathscr{G}} Y \mid \mathbf{W}$ would denote that X is d-separated from Y conditional on $\mathbf{W}$ [Pea09].

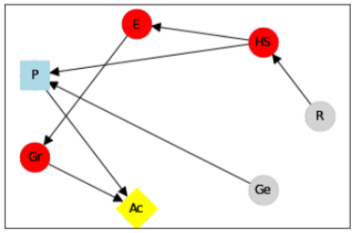

*Fig. 4: A CID for the **Extended Grade Prediction** example with the variables that admit VoI in a darker colour, red (plotted using* PyCID*).*

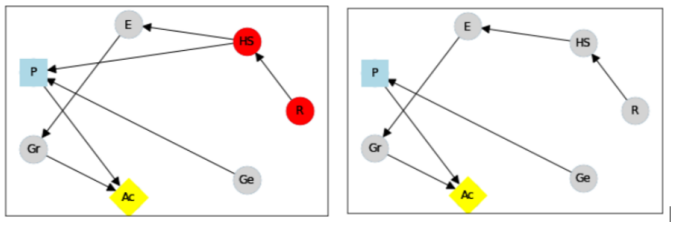

*Fig. 5: (Left) The original CID for the **Extended Grade Prediction** example with the variables that admit an RI in a darker colour, red, and (Right) the modified CID in which now no node admits an RI (plotted using* PyCID*).*

fairness of AI systems. A variable admits an (RI) if a decision-maker benefits from making its decision causally responsive to the variable [ECL+21][6].

RI Graphical Criterion: A single decision CID, $\mathcal{G}$, admits a **response incentive** on $X \in \mathbf{X}$ if and only if there is a directed path $X \dashrightarrow D$ in the requisite graph[7] $\mathcal{G}_{req}$ where $\mathcal{G}_{req}$ is the result of removing from $\mathcal{G}$ all information links from non-requisite observations.

To demonstrate how to find the nodes which admit RIs, we will again consider the **Extended Grade Prediction** example. As we did with VoI, we can list all of the nodes that admit RIs in the CID (line 1) or we can display the result visually (line 2) with the result shown in Figure 5 (Left):

```
1    pycid.admits_ri_list(cid, 'P')
2    cid.draw_property(lambda node:
3          pycid.admits_ri(cid, 'P', node))
```

Implementing CIDs in *PyCID* can help suggest how to improve the fairness of AI systems because [ECL+21] argue that an RI on a sensitive attribute can be interpreted as problematic from a fairness perspective. A decision is considered counterfactually unfair if a change to a sensitive attribute, such as race or gender, would change the decision [KLRS17]. Therefore, an RI on a sensitive attribute indicates that counterfactual unfairness is incentivised; specifically, it implies that all optimal policies are counterfactually unfair. To mitigate this, [ECL+21] propose redesigning the grade-predictor. By removing the predictor's access to knowledge about the student's high school (i.e., the edge $HS \rightarrow P$), there will no longer be an RI on a sensitive attribute. The following code trims the edge and shows that now no node admits an RI in the modified CID (Figure 5 (Right)):

```
cid.remove_edge('HS', 'P')
cid.draw_property(lambda node: \
      pycid.admits_ri(cid, 'P', node))
```

Value of Control (VoC) and Instrumental Control Incentives (ICI)

We now turn to Value of Control (VoC) and Instrumental Control Incentives (ICI) and show that implementing the latter in *PyCID* can help design safer AI systems. Intuitively, a variable has *positive value of control (VoC)* if a decision-maker could benefit from choosing that variable's value.

VoC Definition: For a CIM $\mathcal{M}$, the **value of control** for a non-decision node $X \in \mathbf{V} \setminus \mathbf{D}$ is $\max_{\pi} \max_{g^X} \mathcal{U}^i_{\mathcal{M}_{g^X}}(\pi) - \max_{\pi} \mathcal{U}^i_{\mathcal{M}}(\pi)$. $\mathcal{M}_{g^X}$ denotes the CIM $\mathcal{M}$ after intervening on X with any CPD, g^X, that respects the graph.

6. For a formal definition, we refer the reader to [ECL+21].

7. A requisite graph is also known as a minimal reduction, trimmed_graph, or d-reduction.

VoC Graphical Criterion: A single decision CID, $\mathcal{G}$, admits **positive value of control** for a node $X \in \mathbf{V} \setminus \{D\}$ if and only if there is a directed path $X \dashrightarrow U$ in the requisite graph $\mathcal{G}_{req}$.

Although VoC is a useful concept, it does not consider whether it is actually possible for an agent to control that variable. Therefore, [ECL+21] introduce Instrumental Control Incentives, which can be intuitively understood as follows: if the agent got to choose D to influence X independently of how D influences other aspects of the environment, would that choice matter? In other words, is controlling X instrumentally useful for maximising utility? The graphical criteria for ICI in a single-decision CID is:

ICI Graphical Criterion: A single decision CID, $\mathcal{G}$, admits an **instrumental control incentive** on $X \in \mathbf{V}$ if and only if $\mathcal{G}$ has a directed path from the decision D to a utility node $U \in \mathbf{U}$ that passes through X.

To demonstrate how to find these incentives in *PyCID*, we introduce another example from [ECL+21].

Content recommendation: An AI algorithm has the task of choosing posts (P) to show a user, to maximise the user's click rate (C). The designers want the algorithm to present content adapted to each user's original opinions (O) to optimize clicks; the algorithm does not know the user's true original opinions, so it instead relies on an approximate model (M). However, the designers are worried that the algorithm will use polarising content to influence user opinions (I) so that the user clicks more predictably:

```
cid = pycid.CID(
    [
        ("O", "M"),
        ("O", "I"),
        ("M", "P"),
        ("P", "I"),
        ("I", "C"),
        ("P", "C"),
    ],
    decisions=["P"],
    utilities=["C"],
)

cid.draw_property(lambda node: \
      pycid.admits_ici(cid, 'P', node))
```

With RI, we showed that implementing CIDs in *PyCID* can aid the design of fairer systems; with ICI, we demonstrate how *PyCID* can be used to help design safer AI systems. First, we can use analogous functions to what we used for VoI and RI - *pycid.admits_voc_list(cid)* and *pycid.admits_ici_list(cid, 'P')* - to find that O, M, I, and C can have positive VoC whilst I, P, and C admit ICI. From this, because I (influenced user opinions) admits an instrumental control incentive, we discover that the content recommender may seek to influence that variable to attain utility.

[ECL+21] offer an alternative content recommender design that avoids this undesirable behaviour. Instead of being rewarded for the true click-through rate, the content recommender is rewarded for the clicks it would be predicted to have, based on a separately trained model of the user's preferences. The modified CID for this changed model is shown in Figure 6 c) where the old utility node *C* (actual clicks) has become *PC* (predicted clicks):

```
cid = pycid.CID(
    [
        ("O", "M"),
        ("O", "I"),
        ("M", "P"),
        ("M", "PC"),
        ("P", "I"),
        ("P", "PC"),
    ],
    decisions=["P"],
    utilities=["PC"],
)

cid.draw_property(lambda node: \
      pycid.admits_ici(cid, 'P', node))
```

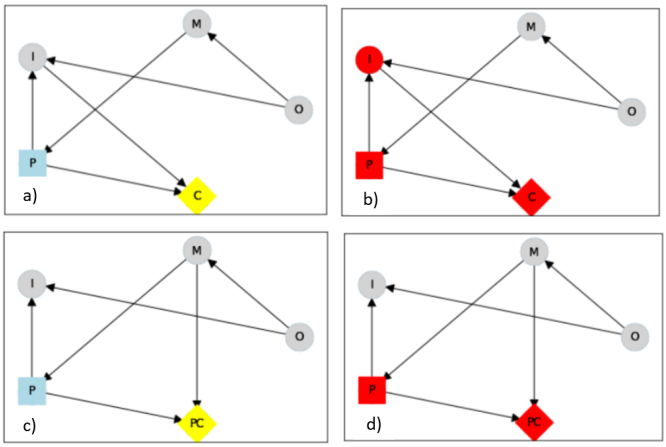

Fig. 6: *The original CID for the* ***Content recommendation*** *example in (a) with (b) the variables that admit ICI in a darker colour, red, and (c) the modified content recommender's CID in which (d) I no longer admits an ICI (plotted using* PyCID*).*

Multi-agent (Causal) Influence Diagrams

In this section, we will show how to instantiate MA(C)IDs/MA(C)IMs in *PyCID* and demonstrate a selection of methods for analysing games (strategic interactions between self-interested players) including strategic relevance [KM03] and finding Nash equilibria (NE) [N+50].

Recall from the *Background* section that a Multi-agent Causal Influence Diagram/Model (MACID/MACIM) is a simple multi-agent extension of a CID/CIM [HFE+]. For our purpose, all that's important is that there is now a set of $\mathbf{N}$ agents and so the decision and utility nodes are partitioned into $\{\mathbf{D}^i\}_{i\in\mathbf{N}}$ and $\{\mathbf{U}^i\}_{i\in\mathbf{N}}$ to correspond to their association with a particular agent $i \in \mathbf{N}$. We also again underline that the only difference between statistical multi-agent influence diagrams/models (MAIDs/MAIMs) and MACIDs/MACIMs is that the edges represent every causal relationship between the random variables chosen to be endogenous variables in the model, as opposed to just associational relationships. Nevertheless, because MACIDs subsume MAIDs (in the sense of Pearl's *causal hierarchy* [Pea09]), everything we can do in a MAID, we can also do in a MACID. Therefore, for the two examples we present here, MAIDs and MACIDs can be viewed as the same.

To serve as our example, we shall use the Prisoner's Dilemma, which is probably the best known simultaneous and symmetric two-player game:

Prisoner's Dilemma: Two prisoners, suspected of committing a robbery together, are isolated and urged to confess. Each is concerned only with getting the shortest possible prison sentence for himself and must decide whether to confess without knowing his partner's decision. Both prisoners, however, know the consequences of their decisions. Each year spent in prison can be represented as -1 utility and so the payoff matrix for this game (or Normal form) is given in Figure 7.

		D^2	
		cooperate	defect
D^1	cooperate	-1,-1	-3,0
	defect	0,-3	-2,-2

Fig. 7: *Normal form game giving the payoffs for each player in the* ***Prisoner's Dilemma****. Player 1 (2) is the row (column) player.*

MA(C)IDs and MA(C)IMs are instantiated as `MACID` objects with identical syntax to `CID` objects except for there being multiple agents and so we can draw them in the same way. Figure 8 (Left) shows that in *PyCID*, consistent with (C)IDs, decision nodes are drawn as rectangles and utility nodes are drawn as diamonds; however, because we now have more than one player, we reserve colouring to denote agent membership: each agent is assigned a unique colour. Chance nodes remain as grey circle (Figure 11):

```
macid = pycid.MACID(
    [
        ("D1", "U1"),
        ("D1", "U2"),
        ("D2", "U1"),
        ("D2", "U2"),
    ],
    # specifies each agent's decision and utility nodes.
    agent_decisions={1: ['D1'], 2: ['D2']},
    agent_utilities={1: ['U1'], 2: ['U2']},
)

d1_dom = ['c', 'd']
d2_dom = ['c', 'd']

agent1_payoff = np.array([[-1, -3], [0, -2]])
agent2_payoff = np.transpose(agent1_payoff)

macid.add_cpds(
    D1=d1_dom,
    D2=d2_dom,
    U1=lambda d1, d2: agent1_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)],
    U2=lambda d1, d2: agent2_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)]
)
macid.draw()
```

The following command tells us that the second player (agent) receives expected utility = -3 (i.e., they will spend 3 years in prison) given that player 1 decides to defect and player 2 decides to cooperate. This agrees with the payoff matrix in Figure 7:

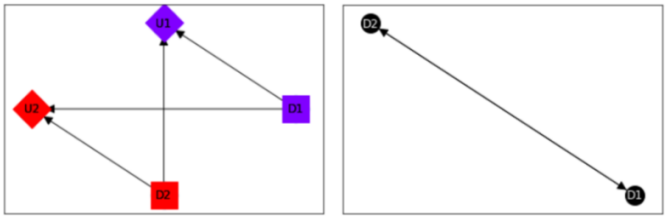

Fig. 8: A MACID for the ***Prisoner's Dilemma*** *(Left) and its corresponding relevance graph (Right) (plotted using* PyCID*).*

```
macid.expected_utility(context={'D1':'d', 'D2': 'c'},
                       agent=2)
```

Strategic relevance is a useful concept for analysing decisions made in games; it asks which other decisions' decision rules need to be already be known before we can optimise a particular decision rule. [KM03] introduced the graphical criterion *s-reachability* for determining this from the graph:

S-reachability Graphical Criterion: Another decision node D' is **s-reachable** from a decision $D \in \mathbf{D}^i$ in a MA(C)ID, $\mathscr{M} = (\mathbf{N}, \mathbf{V}, \mathbf{E})$, if a newly added parent $\hat{D}'$ of D' satisfies $\hat{D}' \not\perp_{\mathscr{G}} \mathbf{U}^i \cap \mathbf{Desc}_D \mid \mathbf{Fa}_D$.

Using *PyCID*, lines 1 and 2 below evaluate to *True*, which tells us that each decision strategically relies on the other; each prisoner would be better off knowing the other prisoner's policy before deciding on their own action. To show this visually, line 3 plots the MACID's relevance graph [KM03] (Figure 8 Right):

```
1  macid.is_r_reachable('D1', 'D2')
2  macid.is_r_reachable('D2', 'D1')
3  pycid.RelevanceGraph(macid).draw()
```

We now turn to finding NE in games. We use $\pi_{\mathbf{A}}$ to denote player i's set of decision rules for decisions $\mathbf{A} \subseteq \mathbf{D}^i$, given a partial policy profile $\pi_{-\mathbf{A}}$ over all of the other decision nodes in a MA(C)ID, $\mathscr{M}$. We write $\mathscr{U}^i_{\mathscr{M}}(\pi_{\mathbf{A}}, \pi_{-\mathbf{A}})$ to denote the expected utility for player i under the policy profile $\pi = (\pi_{\mathbf{A}}, \pi_{-\mathbf{A}})$.

Definition: [KM03] A full policy profile π is a **Nash equilibrium (NE)** in a MA(C)IM $\mathscr{M}$ if, for every player $i \in \mathbf{N}$, $\mathscr{U}^i_{\mathscr{M}}(\pi^i, \pi^{-i}) \geq \mathscr{U}^i_{\mathscr{M}}(\hat{\pi}^i, \pi^{-i})$ for all $\hat{\pi}^i \in \Pi^i$.

To find all pure NE in the MA(C)IM corresponding to the **Prisoner's Dilemma**:

```
macid.get_all_pure_ne()
```

This method returns a list of all pure NE in the MA(C)ID. Each NE comes as a list of `StochasticFunctionCPD` objects, one for each decision node in the MA(C)ID:

```
[[StochasticFunctionCPD<D1>
   {}  ->  d,
StochasticFunctionCPD<D2>
   {}  ->  d]]
```

In the **Prisoner's Dilemma**, there is only one NE and this involves both players defecting. We can then find that the expected utility for each agent is -2 under this NE joint policy profile:

```
all_pure_ne = macid.get_all_pure_ne()
macid.add_cpds(*all_pure_ne[0])
macid.expected_utility({}, agent=1)
macid.expected_utility({}, agent=2)
```

PyCID can also be used to find subgame perfect equilibria (SPE) [Sel65]. A SPE is a NE where no player makes a *non-credible threat* - an action that, if the player is rational, they would never actually carry out.

Definition: [HFE+21] A full policy profile π is a **subgame perfect equilibrium (SPE)** in a MA(C)IM $\mathscr{M}$ if π is an NE in every MAIM subgame[8] of $\mathscr{M}$.

The **Prisoner's Dilemma** MAIM has no proper MAIM subgames and so the NE we found above is (trivially) also a SPE. Therefore, to demonstrate how *PyCID* distinguishes between NE and SPE, we use the following example:

Taxi Competition: Two autonomous taxis, operated by different companies, are driving along a road with two hotels located next to one another - one expensive and one cheap. Each taxi must decide (one first, then the other) which hotel to stop in front of, knowing that it will likely receive a higher tip from guests of the expensive hotel. However, if both taxis choose the same location, this will reduce each taxi's chance of being chosen by that hotel's guests. The payoffs for each player are shown in Figure 9 and the MACIM for this example is instantiated in *PyCID* below

		D^2	
	U^1	expensive	cheap
D^1	expensive	2	5
	cheap	3	1

		D^2	
	U^2	expensive	cheap
D^1	expensive	2	5
	cheap	3	1

Fig. 9: Payoff matrices for taxi 1 (left) and taxi 2 (right) for the ***Taxi Competition****.*

```
macid = MACID(
    [("D1", "D2"), ("D1", "U1"), ("D1", "U2"),
    ("D2", "U2"), ("D2", "U1")],
    agent_decisions={1: ["D1"], 2: ["D2"]},
    agent_utilities={1: ["U1"], 2: ["U2"]},
)

d1_dom = ["e", "c"]
d2_dom = ["e", "c"]
agent1_payoff = np.array([[2, 5], [3, 1]])
agent2_payoff = agent1_payoff.T

macid.add_cpds(
    D1=d1_dom,
    D2=d2_dom,
    U1=lambda d1, d2: agent1_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)],
    U2=lambda d1, d2: agent2_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)],
)
```

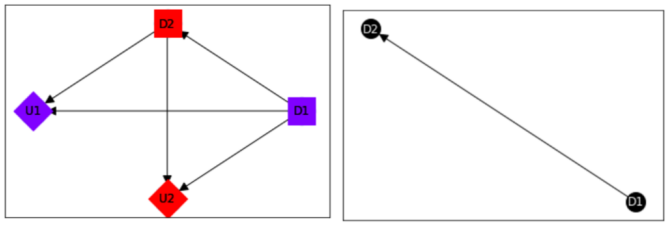

Fig. 10: A MA(C)ID for the ***Taxi Competition*** *and its corresponding relevance graph (plotted using* PyCID*).*

This MA(C)IM has three pure NE, which are found using *macid.get_all_pure_ne()*. We can also find the decision nodes in

8. We refer the interested reader to [HFE+21] for a definition of a MAIM subgame.

each MAID subgame (see [HFE+21]), the decision nodes that can be optimised independently from the rest:

```
macid.decs_in_each_maid_subgame()
[{'D2'}, {'D1', 'D2'}]
```

We can find the NE in the only proper subgame:

```
macid.get_all_pure_ne_in_sg(decisions_in_sg=['D2'])
```

and finally all SPE in the MA(C)IM. The **Taxi Competition**'s MACIM has only one pure SPE:

```
macid.get_all_pure_spe()

[[StochasticFunctionCPD<D2>
   {'d1': 'c'}  ->  e
   {'d1': 'e'}  ->  c,
StochasticFunctionCPD<D1>
   {}  ->  e]]
```

Random (C)IDs and MA(C)IDs

PyCID has other features that can be useful for researchers. In particular, the library contains functions for instantiating random (MA)(C)IDs. This is useful for estimating the average properties of graphs, or for finding a counterexample to some conjecture. The first example below finds and plots a random 10-node, single-agent (C)ID with two decision nodes and three utility nodes. The second example finds and plots a random 12-node MA(C)ID with two agents. The first agent has one decision and two utility nodes, the second agent has three decisions and two utility nodes. In both these examples, we set the *add_cpds* flag to *False* to create non-parameterised (MA)(C)IDs. If one sets this flag to *True*, each chance and utility node is assigned a random CPD, and each decision node a domain to instantiate a (MA)CIM. One can also force every agent in the (MA)(C)ID to have sufficient recall; an agent has sufficient recall if the relevance graph restricted to include just that agent's decision nodes is acyclic. This can be useful for certain incentives analyses [vMCE]. The *edge_density* and *max_in_degree* parameters set the density of edges in the (MA)(C)ID's DAG as a proportion of the maximum possible number $(n \times (n-1)/2)$ and the maximum number of edges incident to a node in the DAG. To find a (MA)(C)ID that meets all of the specified constraints, *PyCID* uses rejection sampling and so *max_resampling_attempts* specifies the number of samples to try before timing out:

```
cid = pycid.random_cid(
    number_of_nodes=10,
    number_of_decisions=2,
    number_of_utilities=3,
    add_cpds=False,
    sufficient_recall=False,
    edge_density=0.4,
    max_in_degree=5,
    max_resampling_attempts=100,
)
cid.draw()

macid = pycid.random_macid(
    number_of_nodes=12,
    agent_decisions_num=(1, 3),
    agent_utilities_num=(2, 2),
    add_cpds=False,
    sufficient_recall=False,
    edge_density=0.4,
    max_in_degree=5,
    max_resampling_attempts=500,
)
macid.draw()
```

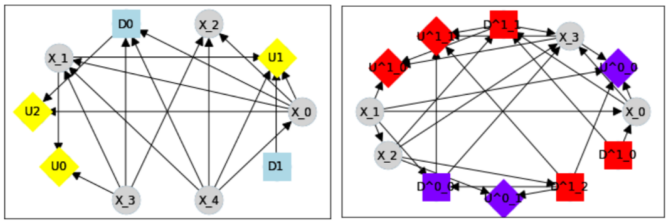

Fig. 11: *A random (C)ID and MA(C)ID created in* PyCID.

Conclusions and Future Directions

PyCID is a Python library for solving and analysing single-agent and multi-agent (causal) influence diagrams. Several key classes - `CausalBayesianNetwork`, `CID`, and `MACID` - enable decision problems to be solved and the effects of causal interventions to be studied whilst *PyCID*'s analysis functions can find graphical properties such as incentives in CIDs and reasoning patterns in MACIDs. This makes *PyCID* a customizable, but powerful library for testing research ideas and exploring applications. Moreover, implementing examples programmatically can substantiate the claims made by ID researchers about the benefit of their work; one can assess how different quantities vary over the parameter space or empirically verify complexity results [HFE+]. Single-agent and multi-agent (causal) influence diagrams are an area of active research, so as theory develops, the *PyCID* library will also grow. Extensions will likely include:

- Support for finding incentives in multi-decision CIDs [vMCE].
- Support for Structural Causal Models [Pea09] and therefore also quantitative RI and ICI.
- More game-theoretic concepts (e.g. more equilibrium concepts).
- Support for multi-agent incentives.

In this paper, we have demonstrated the usefulness of *PyCID* by focusing on causal influence diagrams; however, this library is also well suited for working with statistical influence diagrams. The development team would like to invite researchers from any domain to use *PyCID* to test the package for diverse applications, to contribute new methods and functions, and to join our Causal Incentives Working Group: https://causalincentives.com/. The *PyCID* repository is available on GitHub under our working group's organization: https://github.com/causalincentives/pycid.

Acknowledgements

Fox acknowledges the support of the EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (Reference: EP/S024050/1).

References

[AP15] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic Graphical Models using Python. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 6–11, 2015. doi:10.25080/Majora-7b98e3ed-001.

[Bay] BayesFusionLLC. SMILE: Structural Modeling, Inference, and Learning Engine. URL: https://www.bayesfusion.com/smile.

[BP16] Emanuele Borgonovo and Elmar Plischke. Sensitivity analysis: a review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016. doi:10.1016/j.ejor.2015.06.032.

[CLEL20] Ryan Carey, Eric Langlois, Tom Everitt, and Shane Legg. The incentives that shape behaviour. *arXiv preprint arXiv:2001.07118*, 2020.

[CVH20] Michael Cohen, Badri Vellambi, and Marcus Hutter. Asymptotically unambitious artificial general intelligence. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 2467–2476, 2020. doi:10.1609/aaai.v34i03.5628.

[DBDSMW20] Gaspard Ducamp, Philippe Bonnard, Christian De Sainte Marie, and Pierre-Henri Wuillemin. aGrUM/pyAgrum : a Toolbox to Build Models and Algorithms for Probabilistic Graphical Models in Python. In *10th International Conference on Probabilistic Graphical Models*, volume 138 of *Proceedings of Machine Learning Research*, pages 173–184, Skørping, Denmark, 2020. URL: https://hal.archives-ouvertes.fr/hal-02911619.

[ECL+21] Tom Everitt, Ryan Carey, Eric Langlois, Pedro Ortega, and Shane Legg. Agent incentives: A causal perspective. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence,(AAAI-21). Virtual. Forthcoming*, 2021.

[EHKK21] Tom Everitt, Marcus Hutter, Ramana Kumar, and Victoria Krakovna. Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective. *Synthese*, pages 1–33, 2021. doi:10.1007/s11229-021-03141-4.

[EKKL19] Tom Everitt, Ramana Kumar, Victoria Krakovna, and Shane Legg. Modeling AGI safety frameworks with causal influence diagrams. In *IJCAI AI Safety Workshop*, 2019. arXiv:1906.08663.

[Góm04] Manuel Gómez. Real-world applications of influence diagrams. In *Advances in Bayesian networks*, pages 161–180. Springer, 2004. doi:10.1007/978-3-540-39879-0_9.

[HFE+] Lewis Hammond, James Fox, Tom Everitt, Ryan Carey, Alessandro Abate, and Michael Wooldridge. Causality in games. *Forthcoming*.

[HFE+21] Lewis Hammond, James Fox, Tom Everitt, Alessandro Abate, and Michael Wooldridge. Equilibrium refinements for multi-agent influence diagrams: Theory and practice. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 574–582, 2021.

[HM05] Ronald A Howard and James E Matheson. Influence diagrams. *Decision Analysis*, 2(3):127–143, 2005. doi:10.1287/deca.1050.0020.

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

[Hol20] Koen Holtman. Towards AGI Agent Safety by Iteratively Improving the Utility Function. In Ben Goertzel, Aleksandr I Panov, Alexey Potapov, and Roman Yampolskiy, editors, *Artificial General Intelligence*, pages 205–215, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-52152-3_21.

[How66] Ronald A Howard. Information value theory. *IEEE Transactions on systems science and cybernetics*, 2(1):22–26, 1966.

[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.

[Hun07] John D Hunter. Matplotlib: A 2D graphics environment. *IEEE Annals of the History of Computing*, 9(03):90–95, 2007. doi:10.1109/MCSE.2007.55.

[KLRS17] Matt Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. Counterfactual fairness. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 4067–4077. Neural information processing systems foundation, 2017. URL: http://arxiv.org/abs/1703.06856, arXiv:1703.06856.

[KM03] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. *Games and economic behavior*, 45(1):181–221, 2003. doi:10.1016/s0899-8256(02)00544-4.

[KM08] Uffe B Kjaerulff and Anders L Madsen. Bayesian networks and influence diagrams. *Springer Science+ Business Media*, 200:114, 2008.

[LE21] Eric Langlois and Tom Everitt. How RL agents behave when their actions are modified. In *AAAI*, 2021. arXiv:2102.07716.

[MIMH+76] Allen C Miller III, Miley W Merkhofer, Ronald A Howard, James E Matheson, and Thomas R Rice. Development of automated aids for decision analysis. Technical report, STANFORD RESEARCH INST MENLO PARK CA, 1976. doi:10.21236/ada026379.

[N+50] John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950. doi:10.2307/j.ctv173f1fh.6.

[Pea09] Judea Pearl. *Causality*. Cambridge university press, 2009.

[PG07] Avi Pfeffer and Yakov Gal. On the reasoning patterns of agents in games. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 102–109. AAAI Press, 2007. 22nd national conference on Artificial intelligence, AAAI-07 ; Conference date: 22-07-2007 Through 26-07-2007. URL: https://www.aaai.org/Conferences/AAAI/aaai07.php.

[Sel65] Reinhard Selten. Spieltheoretische behandlung eines oligopolmodells mit nachfrageträgheit: Teil i: Bestimmung des dynamischen preisgleichgewichts. *Zeitschrift für die gesamte Staatswissenschaft/Journal of Institutional and Theoretical Economics*, (H. 2):301–324, 1965.

[Sha86] Ross D Shachter. Evaluating influence diagrams. *Operations research*, 34(6):871–882, 1986.

[vMCE] Chris van Merwijk, Ryan Carey, and Tom Everitt. Techniques for analysing multi-decision influence diagrams. *Forthcoming*.

Social Media Analysis using Natural Language Processing Techniques

Jyotika Singh[‡*]

Abstract—Social media is very popularly used every day with daily content viewing and/or posting that in turn influences people around this world in a variety of ways. Social media platforms, such as YouTube, have a lot of activity that goes on every day in terms of video posting, watching and commenting. While we can open the YouTube app on our phones and look at videos and what people are commenting, it only gives us a limited view as to kind of things others around us care about and what is trending amongst other consumers of our favorite topics or videos. Crawling some of this raw data and performing analysis on it using Natural Language Processing (NLP) can be tricky given the different styles of language usage by people in today's world. This effort highlights the YouTube's open Data API and how to use it in python to get the raw data, data cleaning using NLP tricks and Machine Learning in python for social media interactions, and extraction of trends and key influential factors from this data in an automated fashion. All these steps towards trend analysis are discussed and demonstrated with examples that use different open-source python tools.

Index Terms—nlp, natural language processing, social media data, youtube, named entitity recognition, ner, keyphrase extraction

Introduction

Social media has large amounts of activity every second across the globe. Analyzing text similar to text coming from a social media data source can be tricky due to the absence of writing style rules and norms. Since this kind of data entails user written text from a diverse set of locations, writing styles, languages and topics, it is difficult to normalize data cleaning, extraction, and Natural Language Processing (NLP) methods.

Social media data can be extracted using some official and open APIs. Examples of such APIs include YouTube Data API and Twitter API. One important thing to note would be to ensure one's use case fits within compliance of API guidelines. In this effort, the YouTube Data API will be discussed along with common gotchas and useful tools that can be leveraged to access data.

One can perform NLP if the text data type is available for analysis. The nature of noise seen in text from social media sources will be discussed and presented. Cleaning of the noisy text using python techniques and open-source packages will be further analyzed. Social media data additionally entails statistics of content popularity, likes, dislikes and more. Analysis on statistical and text extracted from YouTube API will be discussed and evaluated.

Finally, trend analysis will be performed using open-source python tools, social media data, statistics, NLP techniques for data cleaning and named entity recognition (NER) for a story-telling analytics piece.

* *Corresponding author: singhjyotika811@gmail.com*
‡ *ICX Media, Inc.*

Natural Language Processing

Natural language processing (NLP) is the computer manipulation of natural language. Natural language refers to language coming from a human, either written or spoken. [Wik21] defined NLP as follows: NLP is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. The result is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. At one extreme, it could be as simple as counting word frequencies to compare different writing styles. [BKL09] mentions, "At the other extreme, NLP involves "understanding" complete human utterances, at least to the extent of being able to give useful responses to them. NLP is challenging because Natural language is messy. There are few rules and human language is ever evolving.".

Some of the common NLP tasks on text data include the following.

1) Named entity recognition

   Named-entity recognition (NER) (also known as (named) entity identification, entity chunking, and entity extraction) is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into predefined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc. Some popular Python libraries that can be leveraged to perform named entity recognition for a variety of different entities include SpaCy [HMVLB20] and NLTK [BKL09].

2) Keyphrase extraction

   Keyphrase extraction is the task of automatically selecting a small set of phrases that best describe a given free text document. [BSMH+18] Some popular tools that can be used for keyphrase extraction as

mentioned in this article[1] include Gensim [RS11] and RAKE-NLTK[#]_. Another way keyphrase extraction can be performed is using NLTK [BKL09] methods. This implementation is included in the pyYouTubeAnalysis [Sin21] library.

3) Unigrams/Bigrams/Trigrams analysis

Breaking down text into single words, a pair of consecutive written words or three consecutively written words and analyzing occurrence patterns.

4) Custom classifier building (public dataset -> features -> ML models)

If out-of-box solutions do not exist for one's NLP task, building custom models to help solve for the problem is an option with the help of available data, NLP libraries (such as NLTK[2], SpaCy[3], and gensim[4]), and Machine Learning libraries (scikit-learn[5]).

5) Others

Tokenization, Part-of-speech tagging, Lemmatization & Stemming, Word Sense Disambiguation, Topic modeling, Sentiment Analysis and Text summarization are some other popularly used NLP tasks. This list is not all inclusive.

A human can only see N number of text samples a day to learn, whereas a machine can analyze a lot greater than N. Leveraging machines for NLP tasks along with several processing solutions available with Python, such as multiprocessing[6], can help analyze large amounts of data in a reasonable time-frame.

Potential use cases include the following.

1) Analytics, intelligence and trends

Analyzing patterns in text based on word occurrences, language, combining text occurrences with other available data, topics, sentiment information, NLP method outputs, or combinations thereof.

2) Story telling

Analyzing text using the various NLP techniques along with other statistical and other available data aids in converting raw data to an informative story piece that helps uncover and understand the patterns that exist within the data. Depending on the data available, a time-window analysis can help study patterns as they change with respect to time in terms of word usages, topics, text lengths, or combinations thereof.

1. https://towardsdatascience.com/extracting-keyphrases-from-text-rake-and-gensim-in-python-eefd0fad582f
2. https://pypi.org/project/rake-nltk/
3. https://scikit-learn.org/
4. https://www.nltk.org/
5. https://spacy.io/
6. https://radimrehurek.com/gensim/
7. https://scikit-learn.org/
8. https://docs.python.org/3/library/multiprocessing.html

Social Media APIs

There are several social media platforms that let you programmatically collect publicly available data and/or your own published data via APIs. Whatever you intend to do with this data, it is important to ensure that you use the data in compliance with the API's guidelines and terms and services.

Some types of available requests on YouTube include search, video, channel and comments.

YouTube Data API documentation[7] is a great resource to learn more and get started. At a high level, the getting started[8] steps include registering a project, enabling the project and using the API key generated. With this key, the user can start making requests to the API to crawl data.

Gotchas

There are a few items to keep in mind when using the YouTube Data API. Some of the gotchas while using the api include the following.

1) Rate limits

The API key registered to you comes with a daily quota. The quota-spend depends on the kind of requests you make. API does not warn you in API request response if you are about to finish your daily quota but does throw that error once you have exceeded the daily quota. It is important to know how your application will behave if you hit the quota to avoid unexpected behavior and premature script termination.

2) Error handling

If trying to query for a video, comment or channel that is set to private by the owner, the API throws an error. Your code could end prematurely if you are querying in a loop and one or a few of the requests have that issue. Error handling could help automate one's process better on such expected errors.

Interacting with the YouTube Data API

There are several ways to interact with the YouTube Data API. Some of them are as follows.

1) Use the API web explorer's "Try this API" section[9]
2) Build your own code using API documentation examples[10]
3) Open-source tools

1. Wrappers of YouTube Data API[11] : Libraries that act as wrappers and provide a way to use YouTube Data API V3.

2. pyYouTubeAnalysis :cite *pyYouTubeAnalysis*[12] : This library allows the user to run searches, collect videos and comments, and define search params (search keywords, timeframe, and type). Furthermore, the project includes error handling that allows code execution to continue and not stop due to unforeseen errors while interacting with YouTube data API. Additional features included in pyYouTubeAnalysis are NLP methods for social media text pre-processing mentioned in a later section *Data Cleaning Techniques*, NLTK

9. https://developers.google.com/youtube/v3/docs
10. https://developers.google.com/youtube/v3/getting-started

based keyphrase extraction and SpaCy based Named Entity Recognition (NER) that runs entity extraction on text.

Social Media / YouTube Data Noise

Text fields are available within several places on YouTube, including video title, description, tags, comments, channel title and channel description. Video title, description, tags, and channel title and description are filled by the content/channel owner. Comments on the other hand are made by individuals reacting to a video using words and language.

The challenges in such a data source arise due to writing style diversity, language diversity and topic diversity. Figure 1 shows a few examples of language diversity. On social media, people use abbreviations, and sometimes these abbreviations may not be the most popular ones. Other than the non-traditional abbreviation usage, different languages, different text lengths, and emojis used by commenters are observed.

Data Cleaning Techniques

Based on some noise seen on YouTube and other social media platforms, the following data cleaning techniques have been found to be helpful cleaning methods.

1) Removing URLs

Social media text data comes with a lot of URLs. Depending on the task at hand, removing the urls have been observed to come in handy for cleaning the text. Remove the URLs prior to passing text through keyphrase or NER extractions has been found to return cleaner results. This implementation is also contained in pyYouTubeAnalysis.

```
import re

URL_PATTERN = re.compile(
    r"https?://\S+|www\.\S+",
    re.X
)

def remove_urls(txt):
    """
    Remove urls from input text
    """
    clean_txt = URL_PATTERN.sub(" ", txt)
    return clean_txt
```

2) Removing emojis

Emojis are widely used across social media by users to express emotions. Emoijis provide benefit in some NLP tasks, such as certain sentiment analysis implementations that rely on emoji based detections. On the contrary, for many other NLP tasks, removing emojis from text can be a useful cleaning method that improves the quality of the processed outcome. For named-entity recognition and keyphrase extraction, certain emojis are observed getting falsely detected as locations or nouns of the type NN or NNP. This impacts the quality of the NLP methods. Removing the emojis prior to passing such text through named-entity recognition or keyphrase extractions has been found to return cleaner results. This implementation is also contained in pyYouTubeAnalysis.

```
import re

EMOJI_PATTERN = re.compile(
    "[\U00010000-\U0010ffff]",
    flags=re.UNICODE
)

def remove_emojis(txt):
    """
    Remove emojis from input text
    """
    clean_txt = EMOJI_PATTERN.sub(" ", txt)
    return clean_txt
```

3) Spelling / typo corrections

Some NLP models tend to do very well for a particular style of language and word usage. On social media, the language seen can be accompanied with various incorrectly spelled words, also known as typos. PySpellChecker [LT16][13], Autocorrect[14] and Textblob [Lor18] are examples of open-source tools that can be used for spelling corrections.

4) Language detection and translations

Developing NLP methods on different languages is a challenging and popular problem. Often when one has developed NLP methods for english language text, detection of a foreign language and translation to english serves as a good solution and allows one to keep their NLP methods fixed. Such tasks introduce other challenges such as the quality of language detection and translation. Nonetheless, detection and translation is a popular technique while dealing with multiple different languages. Some examples of Python libraries that can be used for language detection include langdetect [Shu10], Pycld2[15], Textblob [Lor18], and Googletrans[16]. Translate[17] and Googletrans can be used for language translations.

Trend Analysis Case Study

In the year 2020, COVID hit us all hard. The world went through a lot of changes in the matter of no time to reduce the spread of the virus. One such impact was observed massively in the travel and hospitality industry. Figure 2[18] shows the flight search trends between February and November 2020 for domestic and international flight searches from the US using Kayak. Right before lockdown and restrictions were enforced starting in March across different places across the globe, a big spike can be seen in flight searches, correlating with the activity of people trying to fly back home if they were elsewhere before restrictions disabled them to do so.

11. https://developers.google.com/youtube/v3/docs/search/list
12. https://developers.google.com/youtube/v3/quickstart/python
13. https://github.com/rohitkhatri/youtube-python, https://github.com/sns-sdks/python-youtube
14. https://github.com/jsingh811/pyYouTubeAnalysis
15. https://pypi.org/project/pyspellchecker/
16. https://pypi.org/project/autocorrect/
17. https://pypi.org/project/pycld2/
18. https://pypi.org/project/googletrans/
19. https://pypi.org/project/translate/

Fig. 1: *Random sample of YouTube comments representing writing style diversity.*

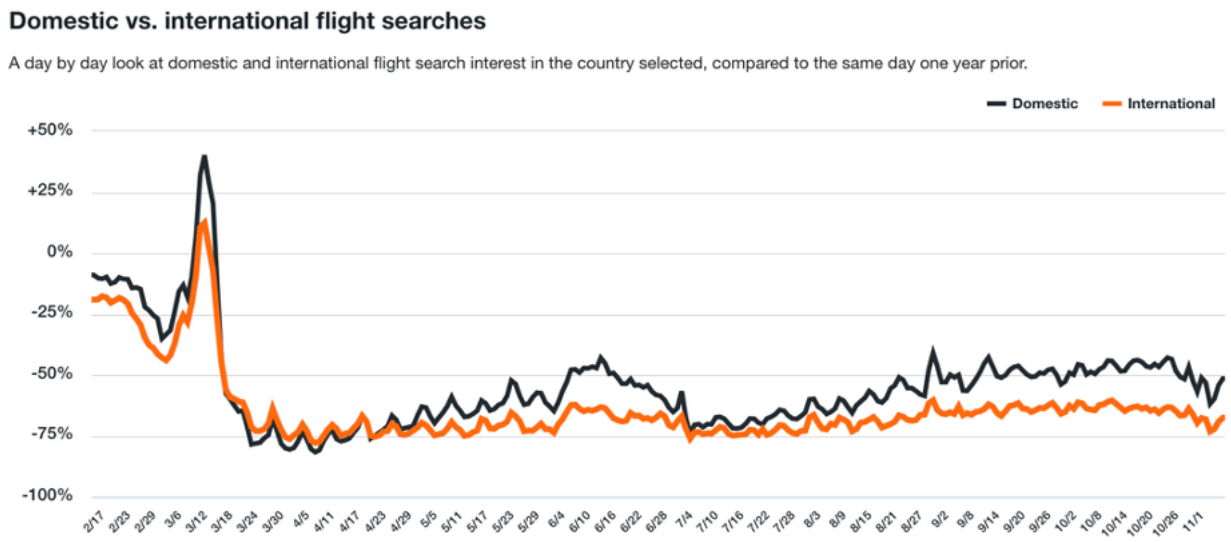

Fig. 2: *Domestic and international flight search patterns in 2020.*

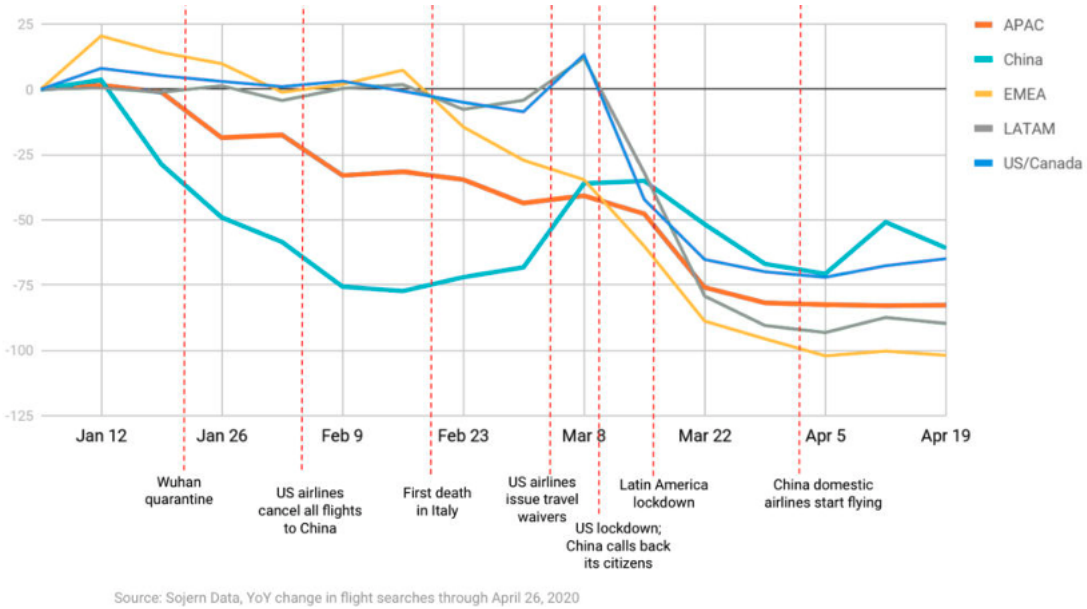

Fig. 3: *Global flight search patterns in 2020.*

A massive reduction in flight searches can further be seen in figure 3[19] showing the impact at a global level. Timeline beyond January of 2020 for China, and beyond March of 2020 for most other locations, faced the most impact as travel was reduced due to COVID imposed events and restrictions.

Aligning with reduced flight searches, reduced hotel search were also reported from March onwards as can be seen in figure 4[20].

Let's try to correlate these findings and understand content consumption within those time periods on YouTube.

First, a search was performed to gather videos about "travel vlogs" using the pyYouTubeAnalysis library. Travel vlogs are a popular content genre on YouTube where a lot of people are able to find reviews, advice and sneak peaks of different destinations that wows them and inspires travel plans. Such videos typically consist of people traveling to different locations and recording themselves at different spots.

Statistically, it can be seen from figures 5, 6 and 7 that travel vlog has been a growing topic of interest and has been growing along with online content consumption over the years up till 2019. A downward trend was seen in average views, comments, and likes on travel vlog videos in 2020, where the views went down by 50% compared to the year before.

To understand the differences between the travel vlog content consumed in 2019 versus 2020 in further detail, a monthly data crawl was performed. Figures 8, 9 and 10 show a month over month comparison between 2019 and 2020 to analyze average audience engagement patterns. The viewership trends reflect the reduction from March onwards when COVID hit most locations across the globe. Figure 11 further shows engagement shift between 2019 and 2020. The trend slopes upwards until March hits, which is when a lot of locations imposed stay at home orders and lockdowns. The trend slopes downwards, picks up a little July onwards, which correlates with the time Europe lifted a lot of the travel restrictions. The chart representing "travel vlog" content engagement largely correlates with the flight search trend as shown in figure 2. It can be seen however, people were still creating travel vlogs and commenting on such videos. Between June and September 2020, amidst a much-reduced travel, what were these videos, what content was getting created, who was creating it, and what were the commenters talking about?

Figure 12 shows a word cloud representation of what these videos talked about generated using keyphrase extraction implementation in pyYouTubeAnalysis, where the text passes through data cleaning techniques prior to keyphrase extraction that is inbuilt within the implementation. Application of these techniques prior to extracting keyphrases eliminated the noisy samples and improved the overall results quality. Additionally, wordcloud [OMIB11][21] was used for creating the visualization. Word cloud is a form of term occurrence visualization where the size of the appearance of a term in the word cloud is directly proportional to its occurrence count. Travel that would entail easier implementation of social distance was seen popping up in 2020, such as hiking, beach trips and road traveling. Location names such as Italy, France and Spain were also seen showing up in the videos.

While we have seen what content gained the most engagement, let's look into who the creators of such content were that drove the most comments and engagement. With the help of engagement statistics and videos read for the 2020 time frame, the YouTube influencer channels that drove high engagement during summer and fall of 2020 include the following.

1) 4K Walk[22] – YouTube channel creating videos about walking tours all over Europe and America.
2) BeachTuber[23] – YouTube channel creating vlogs from different beaches all over Europe.
3) Beach Walk[24] – YouTube channel posting about different beaches all over Europe and America.
4) DesiGirl Traveller[25] – YouTube channel creating videos about India travel.
5) Euro Trotter[26] – YouTube channel creating videos about Europe travel.

A few examples of comments that were being left by audiences of such videos are as follows.

> "i'm going to sorrento in 10 days and i'm so excited. i've been watching tonnes of sorrento and italy vlogs and yours are so lush X) <3"
>
> "Did they require you to have a prior covid test?"
>
> "I loved the tour looked like you guys had fun. im going there next week, how long ago were you there and were there lots of restrictions and closing due to covid"
>
> "Great video man, this place looks amazing. I have never been to Iceland, would love to visit some day. Honestly can't wait for the lockdown to be lifted so I can start travelling again. Thanks for sharing your experience. :)"

It was seen that people expressed interest in inquiring about the lifting of the travel ban due to COVID, pre-travel COVID test requirements, along with the sentiments around being able to travel again. People were seen mentioning a lot of location names in their comments. With the help of named-entity recognition implementation in pyYouTubeAnalysis, location extractions were performed. The underlying process passed the comments through URLs and emojis removal prior to location extraction, which led to cleaner results and reduced manual filtering. Figure 13 shows the location popularly mentioned by commenters in a word cloud representation. One can see European locations, along with some Asian and American locations which correlate with travel restriction reductions in some of the places.

This analysis, including data collection from social media, keyphrase extraction, and NER, was performed using pyYouTube-Analysis library [Sin21][27]. Similar analysis for content other than "travel vlogs" can be performed for custom time windows using similar tools and the other NLP libraries mentioned in this effort.

Conclusion

User content creations and interactions via text on social media platforms contain mixed writing styles, topics, languages, typing

20. https://www.kayak.com/news/category/travel-trends/
21. https://www.sojern.com/blog/covid-19-insights-on-travel-impact-hotel-agency/
22. https://www.sojern.com/blog/covid-19-insights-on-travel-impact-hotel-agency/
23. https://pypi.org/project/wordcloud/, https://www.wordclouds.com/
24. https://youtube.com/c/4KWALK
25. https://youtube.com/c/BeachTuber
26. https://youtube.com/c/BeachWalk
27. https://youtube.com/c/DesiGirlTraveller
28. https://youtube.com/c/EuroTrotter
29. https://github.com/jsingh811/pyYouTubeAnalysis

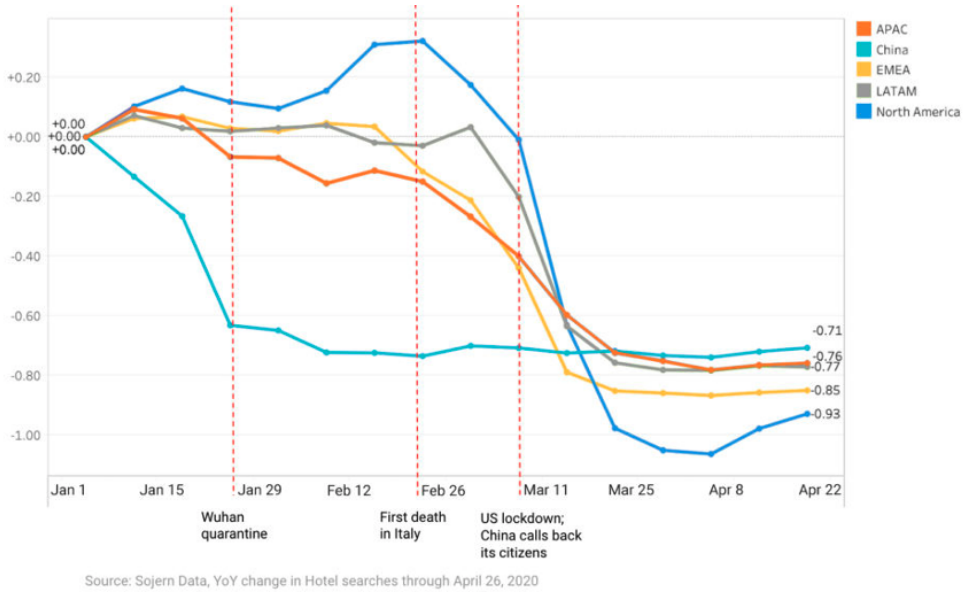

Fig. 4: *Hotel booking search patterns in 2020.*

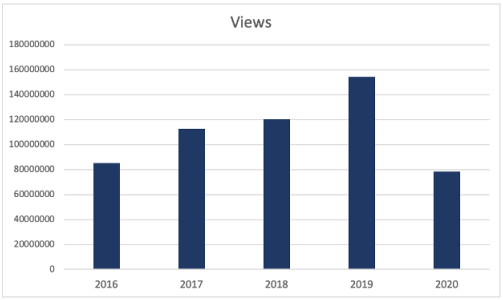

Fig. 5: *Yearly video views.*

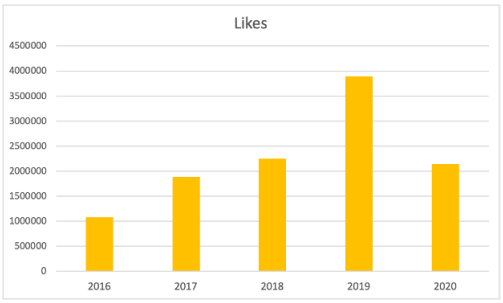

Fig. 6: *Yearly video likes.*

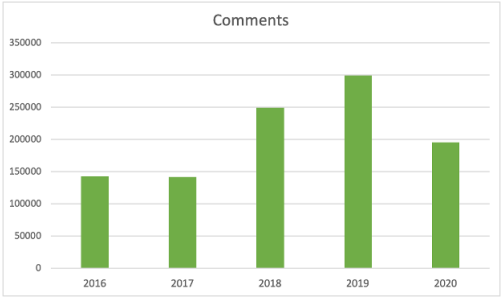

Fig. 7: *Yearly video comments.*

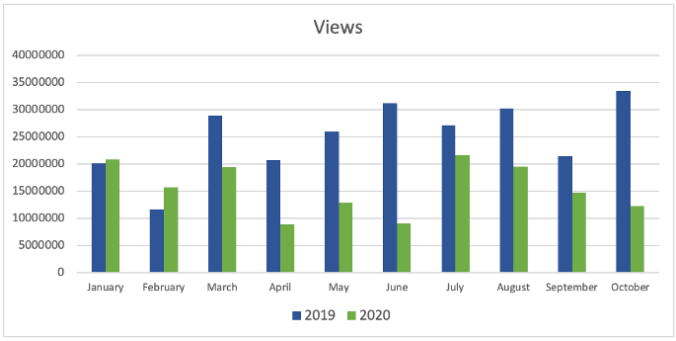

Fig. 8: *Monthly video views for 2019 and 2020.*

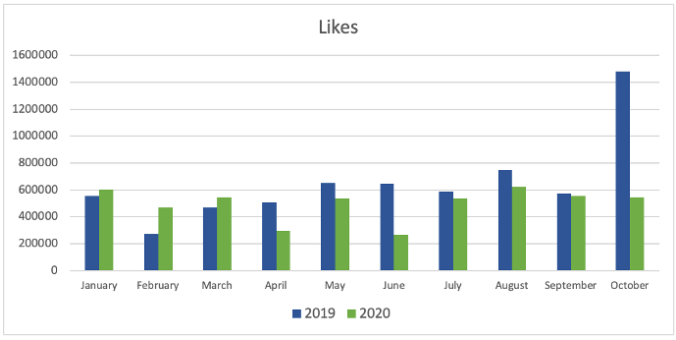

Fig. 9: *Monthly video likes for 2019 and 2020.*

errors, freeform emojis and abbreviations. This diversity of content and language makes it harder to perform NLP tasks on data coming from social media. Described cleaning techniques such as emoji removal, hyperlink removal, language detection and translations, and typo corrections have been found useful in priming and pre-processing language of such nature. Subjecting the text through these methods prior to other Natural Language Processing (NLP) methods such as keyphrase extraction and named-entity recognition result in cleaner output.

Social media data contain statistics in addition to text data

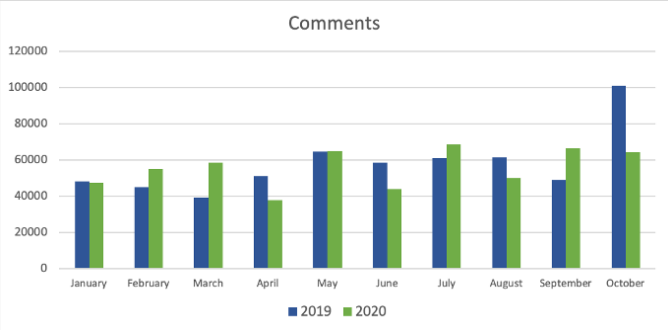

Fig. 10: *Monthly video comments for 2019 and 2020.*

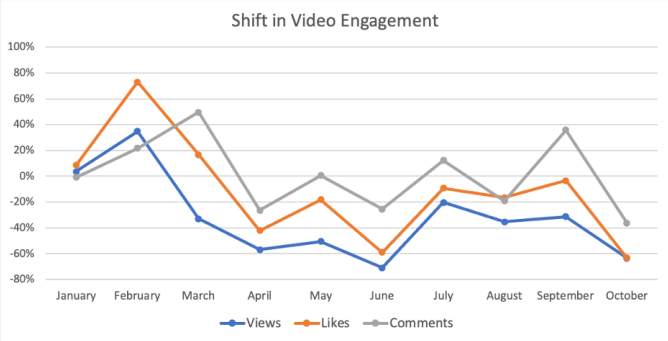

Fig. 11: *Difference in video engagements between 2019 and 2020.*

Fig. 12: *Word cloud of video topics.*

Fig. 13: *Word cloud of location names used in comments.*

that measures human engagement and interest in different types of content. Combining these statistics with inferences from NLP techniques such as named-entity recognition (NER) and keyphrase extraction are found to be helpful in trend analysis, analytics, and observing correlations and affinities of user engagement with social media.

REFERENCES

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. 01 2009.

[BSMH+18] Kamil Bennani-Smires, Claudiu Musat, Andreea Hossmann, Michael Baeriswyl, and Martin Jaggi. Simple unsupervised keyphrase extraction using sentence embeddings. 10 2018. doi:10.18653/v1/K18-1022.

[HMVLB20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020. URL: https://doi.org/10.5281/zenodo.1212303, doi:10.5281/zenodo.1212303.

[Lor18] Steven Loria. textblob documentation. *Release 0.15*, 2, 2018.

[LT16] Pierre Lison and Jörg Tiedemann. Opensubtitles2016: Extracting large parallel corpora from movie and TV subtitles. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016*. European Language Resources Association (ELRA), 2016. URL: http://www.lrec-conf.org/proceedings/lrec2016/summaries/947.html.

[OMIB11] Layla Oesper, Daniele Merico, Ruth Isserlin, and Gary Bader. Wordcloud: A cytoscape plugin to create a visual semantic summary of networks. *Source code for biology and medicine*, 6:7, 04 2011. doi:10.1186/1751-0473-6-7.

[RS11] Radim Rehurek and Petr Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[Shu10] Nakatani Shuyo. Language detection library for java. 2010. URL: http://code.google.com/p/language-detection/.

[Sin21] Jyotika Singh. jsingh811/pyyoutubeanalysis: Youtube data requests and natural language processing on text, 2021. URL: https://zenodo.org/record/5044556, doi:10.5281/ZENODO.5044556.

[Wik21] Wikipedia contributors. Natural language processing — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Natural_language_processing&oldid=1030186679, 2021. [Online; accessed 25-June-2021].

PyBMRB: Data visualization tool for BioMagResBank

Kumaran Baskaran[‡§*], Jonathan R Wedell[‡§], Eldon L. Ulrich[‡§], Jeffery C. Hoch[‡§], John L. Markley[¶‖]

Abstract—The Biological Magnetic Resonance Data Bank (BioMagResBank or BMRB https://bmrb.io), founded in 1988, is the international, open archive for data generated by Nuclear Magnetic Resonance (NMR) spectroscopy of biological systems. NMR spectroscopy is unique among biophysical approaches in its ability to provide a broad range of atomic and higher-level information relevant to the structural, dynamic, and chemical properties of biological macromolecules, as well as report on metabolite and natural product concentrations in complex mixtures and their chemical structures. NMR-STAR is the official data format of BMRB and BMRB provides python parser (PyNMRSTAR https://github.com/uwbmrb/PyNMRSTAR), a data visualization tool (PyBMRB https://github.com/uwbmrb/PyBMRB) and an Application Program Interface (API)(BMRB-API https://github.com/uwbmrb/BMRB-API) to access the BMRB archive. PyBMRB displays the chemical shifts data in each entry as a simulated NMR spectrum and to generates database-wide chemical shift histograms of different atom types in proteins and nucleic acids. PyBMRB provides access to BMRB data through the API and generates portable and interactive visualizations as a single html file. It also supports data visualization workflows using Jupyter Notebooks, which can be both easily created and shared.

Index Terms—NMR Spectroscopy, chemical shifts, proteins, Biological Magnetic Resonance data Bank(BMRB),NMR-STAR, chemical shift histogram, HSQC

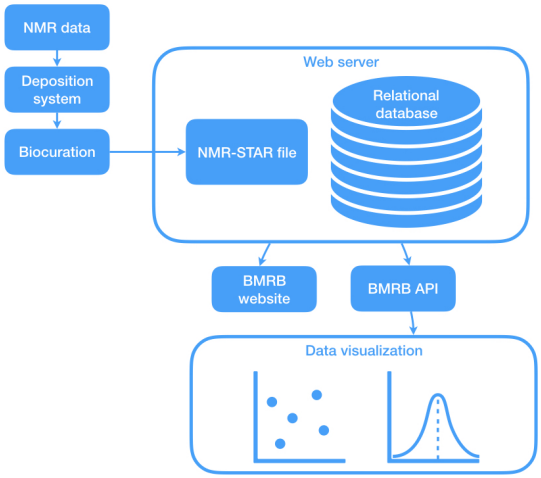

Fig. 1: *BMRB data processing workflow.*

Nuclear Magnetic Resonance (NMR) spectroscopy provides atom-level information relevant to the structural, dynamic, and chemical properties of molecules. The BioMagResBank (BMRB) [UAD+07] provides high-quality, curated NMR spectroscopic data collected from biologically important molecules such as proteins, nucleic acids, carbohydrates, and metabolites and other small compounds. BMRB, which was founded in 1988, became a core member of World Wide Protein Data Bank (wwPDB) [BBK+17] in 2007, and the BMRB Archive became a Core Archive of the wwPDB in 2018. BMRB uses the NMR-STAR [UBD+19] data format to represent experiments, spectral and derived data, and supporting metadata. NMR-STAR is constructed via an object-relational data model using a subset of the Self-defining Text Archival and Retrieval (STAR) specification [HC95]. Following validation and annotation via BMRB's biocuration pipeline (Figure 1), user-deposited data are stored as flat files in NMR-STAR format as well as in a relational database.

To achieve the full power of the BMRB database it is important to be able to retrieve and visualize the data in different scientifically relevant ways. For example, it is much more useful to compare multidimensional NMR data from the same or different BMRB entries in graphical (spectral) format rather than as lists of numerical values in text format. In addition, to understand how chemical shifts of different types of atoms are affected by structural and environmental factors, it is useful to display them as histograms. When browser vendor security policies changed to stop allowing Java Web Applets, BMRB's original visualization tool (DEVise) [LRB+97] written in Java and C++ ceased to function. BMRB originally addressed this by updating DEVise to run as a Java Web Start application. However, in mid-2015 most web browsers stopped supporting Java Web Start and some operating system made it impossible to use without changing operating system security settings.

In response to the demise of DEVise, BMRB developed graphic libraries in Python (PyBMRB) that utilize more modern interactive visualization tools, such as the Plotly visualization tool kit [Inc15] , to reproduce the most commonly used features of DEVise with interactive visualizations. PyBMRB features single-entry (peak position simulation for NMR spectrum) and database-wide visualizations (histograms).

The main motivation behind the project is to provide user friendly access to BMRB data for biologists and biochemists, who find it difficult to understand or utilize the NMR-STAR data model. NMR-STAR is a metadata rich format, which includes all necessary metadata about the NMR sample, sample condition, instrument details, author details and experimental details in addition to the measured chemical shift values. Chemical shifts

* *Corresponding author: baskaran@uchc.edu*
‡ *UCONN Health, Molecular Biology and Biophysics*
§ *263 Farmington Ave. Farmington, CT 06030-3305, USA*
¶ *Department of Biochemistry, University of Wisconsin-Madison*
‖ *433 Babcock Drive, Madison, Wisconsin 53606-1544, USA*

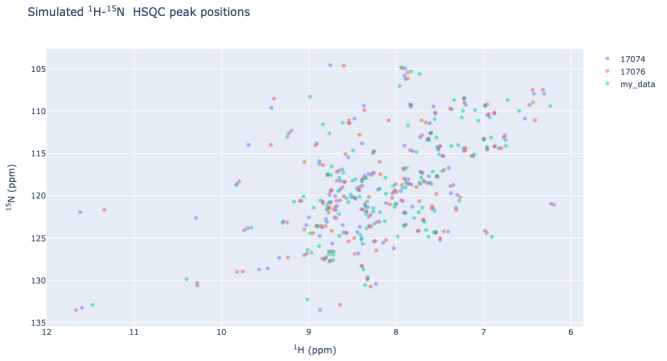

Fig. 2: *Comparison of $^{1}H-^{15}N$ HSQC spectra of arsenate reductase data from user with arsenate reductase entries in the BMRB*

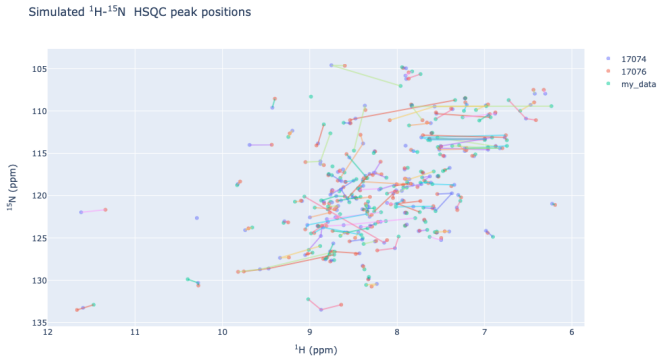

Fig. 3: *The cross peaks in the $^{1}H-^{15}N$ HSQC spectra are connected based on matching sequence order.*

are measured using several multidimensional NMR experiments and expressed one-dimensional assigned chemical shift lists in NMR-STAR data format. Biologists and biochemists prefer to view the chemical shift data graphical spectra rather than as a list of numerical values.

One of the most common and widely used NMR experiments for proteins is the $^{1}H-^{15}N$ Heteronuclear Single Quantum Coherence($^{1}H-^{15}N$ HSQC) [BR80] experiment. This 2D NMR experiment gives cross peaks between nitrogen and hydrogen for each amino acid in the sequence, whose locations strongly depend on the protein three dimensional structure. In spectroscopic perspective the $^{1}H-^{15}N$ HSQC spectrum is considered as the signature or "fingerprint" of the protein. It helps to identify whether the protein sample is in good shape or aggregated and to detect structural changes during ligand biding studies. PyBMRB library generates 2D chemical shift lists by combining the relevant chemical shift values from the given one-dimensional chemical shift list in NMR-STAR format.

The single-entry visualization method can be used, for example, to simulate $^{1}H-^{15}N$ HSQC peak positions from an NMR-STAR file (from one or more specified BMRB entries or from the user's own data) (Figures 2 and 3). It is much easier to detect the chemical shift changes by overlaying multiple $^{1}H-^{15}N$ HSQC rather than by scanning lists of chemical shifts. The most useful feature is that the user may easily compare their NMR measurements with any of the protein of interest in the BMRB database. The Figures 2 and 3 show the comparison of user data with two similar entries from BMRB database. This comparison can be done with the following code

```
from pybmrb import csviz
s=csviz.Spectra()
s.n15hsqc(bmrbid=[17074,17076],
       filename='my_data.str')
```

The chemical shift changes can be traced for each residue by using `groupbyres` option. (Figures 3)

```
s.n15hsqc(bmrbid=[17074,17076],
       filename='my_data.str',
       groupbyres=True)
```

BMRB provides rich chemical shift statistics, which are widely used by NMR spectroscopists and NMR software developers in various ways. The chemical shift histogram of a given atom type help us to understand how strongly it's position depends on the secondary structure elements like alpha helices and beta sheets.

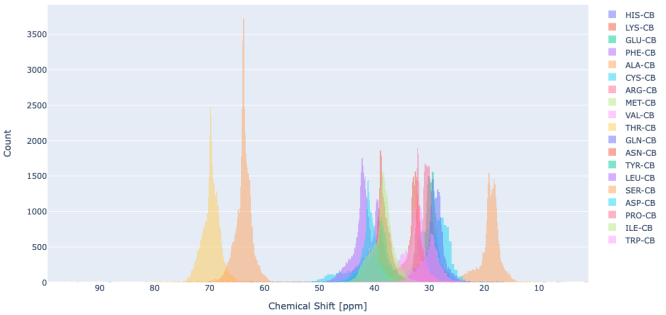

Fig. 4: *Chemical shift distribution of CB atoms in different amino acids.*

These histograms can be easily generated using a simple code using PyBMRB library

```
from pybmrb import csviz
h=csviz.Histogram()
h.hist(atom='CB')
```

Figure 4 shows the comparison of CB chemical shifts for the twenty common amino acids. The chemical shift histogram of a single atom in a given amino acid or list of atoms from different amino acids can be easily generated using PyBMRB.

PyBMRB provides options for filtering data, for example, according to chemical shift ambiguity code(used to describe different types of ambiguous chemical shift assignments https://bmrb.io/software/ambi/) or cutoff values based on standard deviation to exclude outliers. Bond correlation experiments are very common in NMR spectroscopy, and this library can be used to visualize patterns of chemical shift correlations between specified atom types in NMR spectra of proteins or nucleic acids as 2D histograms. For example the chemical shift correlation between Cysteine CB and N is shown in Figure 5.

```
h.hist2d(residue='CYS',atom1='CB',atom2='N')
```

The conditional histogram is another feature, useful during the resonance assignment process to estimate the prior probability for assigning a specific atom number to a peak. The process of labeling each cross peak in the multidimensional NMR spectra by relevant atoms is the most important step in the structure determination process. If the chemical shift values of one or more

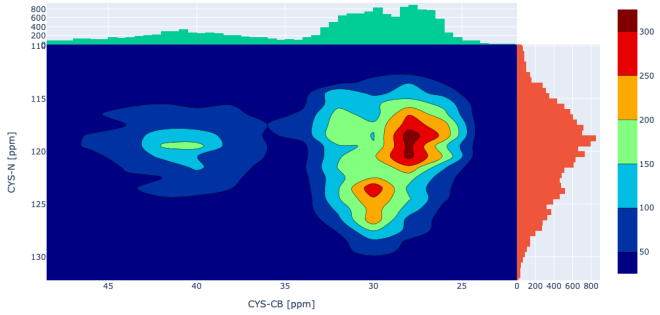

Fig. 5: *Chemical shift correlation of CYS-CB and CYS-N*

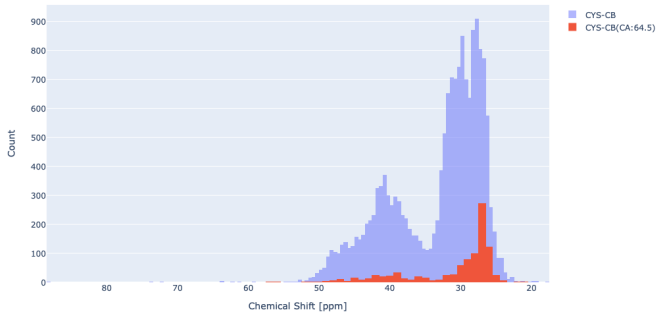

Fig. 6: *Conditional histogram of CYS-CB for CYS-CA=64.5ppm*

atoms for a given amino acid in a protein sequence are known then one can generate the distribution of the chemical shifts of the other atoms in the amino acid using the known chemical shifts as a filter. For example if the chemical shift of CA of Cysteine is known then the distribution of CB chemical shift at the BMRB database can be calculated using the following code

```
h.conditional_hist(residue='CYS',
       atom='CB',
       atomlist=['CA'],
       cslist=[64.5])
```

The overall and the filtered distribution of CYS-CB is shown in Figure 6. The overall bimodal distribution of Cysteine CB indicates that its chemical shifts are strongly depend on secondary structures and for the given value of CA (64.5 ppm) it falls into one of secondary structure element like alpha helix or beta sheet.

The visualizations generated using PyBMRB library are interactive and portable. They can be opened in any modern web browser and zoomed in and out using the mouse. The tooltip will show the peak label and some additional information when hovering over the peak. These visualizations work as a standalone web page, which can be shared via email or website. Since the visualization tools obtain data directly from the BMRB API each time they are generated, there is no need to download or parse the data, and all underlying data are fully up to date. High quality static images can be extracted from the interactive visualizations with a single click and saved or printed.

As a final note, the Jupyter Notebook [KRKP+16] [Com20] is becoming more and more popular among scientists [Per18]. Jupyter is a free, open-source, interactive web tool, known as a computational notebook, that researchers can use to combine software code, computational output, explanatory text and multimedia resources into a single document. PyBMRB can be used in a Jupyter Notebook environment, which enables one to design and document a BMRB data analysis workflow and share it with others. BMRB provides easy access to the PyBMRB library in a Jupyter Notebook environment from its homepage (https://bmrb.io/). This live BMRB Jupyter Notebook was created by using a third party software tool called Binder [PJMBJF+18], which puts PyBMRB and Jupyter Notebook together in a docker container. Examples of BMRB Jupyter Notebooks with access to PyBMRB are available for trial without the need for any installation at https://github.com/uwbmrb/PyBMRB/blob/master/jupyter.md.

BMRB is constantly working to improve the PyBMRB visualization tool. The next update aims to include simulation of more NMR experiment types and include visualization options for other data types such as distance and dihedral-angle restraints that are present in the BMRB database.

BMRB is supported by grant R01GM109046 from NIH/NIGMS.

REFERENCES

[BBK+17] Stephen K Burley, Helen M Berman, Gerard J Kleywegt, John L Markley, Haruki Nakamura, and Sameer Velankar. Protein Data Bank (PDB): The Single Global Macromolecular Structure Archive. *Methods in molecular biology (Clifton, N.J.)*, 1607:627–641, 2017. URL: https://pubmed.ncbi.nlm.nih.gov/28573592https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5823500/, doi:10.1007/978-1-4939-7000-1_26.

[BR80] Geoffrey Bodenhausen and David J. Ruben. Natural abundance nitrogen-15 nmr by enhanced heteronuclear spectroscopy. *Chemical Physics Letters*, 69(1):185–189, 1980. URL: https://www.sciencedirect.com/science/article/pii/0009261480800418, doi:https://doi.org/10.1016/0009-2614(80)80041-8.

[Com20] Executable Books Community. Jupyter book, February 2020. URL: https://doi.org/10.5281/zenodo.4539666, doi:10.5281/zenodo.4539666.

[HC95] S. Hall and A. P. Cook. Star dictionary definition language: Initial specification. *J. Chem. Inf. Comput. Sci.*, 35:819–825, 1995.

[Inc15] Plotly Technologies Inc. Collaborative data science, 2015. URL: https://plot.ly.

[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. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016. URL: https://eprints.soton.ac.uk/403913/, doi:10.3233/978-1-61499-649-1-87.

[LRB+97] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: Integrated querying and visual exploration of large datasets. *SIGMOD Rec.*, 26(2):301–312, June 1997. URL: https://doi.org/10.1145/253262.253335, doi:10.1145/253262.253335.

[Per18] Jeffrey Perkel. Why jupyter is data scientists' computational notebook of choice. *Nature*, 563:145–146, 11 2018. doi:10.1038/d41586-018-07196-1.

[PJMBJF+18] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the*

17th Python in Science Conference, pages 113 – 120, 2018. doi:10.25080/Majora-4af1f417-011.

[UAD+07] Eldon L. Ulrich, Hideo Akutsu, Jurgen F. Doreleijers, Yoko Harano, Yannis E. Ioannidis, Jundong Lin, Miron Livny, Steve Mading, Dimitri Maziuk, Zachary Miller, Eiichi Nakatani, Christopher F. Schulte, David E. Tolmie, R. Kent Wenger, Hongyang Yao, and John L. Markley. BioMagResBank. *Nucleic Acids Research*, 36:D402–D408, 11 2007. URL: https://doi.org/10.1093/nar/gkm957, arXiv:https://academic.oup.com/nar/article-pdf/36/suppl_1/D402/7635401/gkm957.pdf, doi:10.1093/nar/gkm957.

[UBD+19] Eldon L Ulrich, Kumaran Baskaran, Hesam Dashti, Yannis E Ioannidis, Miron Livny, Pedro R Romero, Dimitri Maziuk, Jonathan R Wedell, Hongyang Yao, Hamid R Eghbalnia, Jeffrey C Hoch, and John L Markley. NMR-STAR: comprehensive ontology for representing, archiving and exchanging data from nuclear magnetic resonance spectroscopic experiments. *Journal of Biomolecular NMR*, 73(1):5–9, feb 2019. URL: https://doi.org/10.1007/s10858-018-0220-3, doi:10.1007/s10858-018-0220-3.

Conformal Mappings with SymPy: Towards Python-driven Analytical Modeling in Physics

Zoufiné Lauer-Baré[‡*], Erich Gaertig[‡]

Abstract—This contribution shows how the symbolic computing Python library `SymPy` can be used to improve flow force modeling due to a Couette-type flow, i.e. a flow of viscous fluid in the region between two bodies, where one body is in tangential motion relative to the other. This motion imposes shear stresses on the fluid and leads to a corresponding fluid flow. The flow forces exerted on the moving component are of interest in many applications, for example in system simulations of electrohydraulic valves. There, an eccentrically mounted cylindrical core (the armature) moves within an oil-filled tube (the polecap), experiencing fluid forces due to the viscous oil. `SymPy` can help to understand the range of validity as well as the limitations of analytical relations that are commonly used as standard approximations for these type of forces in many leading system simulation tools. In order to motivate these approaches, this contribution elucidates how the velocity of the flow is determined analytically by solving the Stokes equation in an eccentric annulus with a conformal mapping-approach. Afterwards analytical postprocessing leads to the corresponding flow force. The results obtained with `SymPy` are then checked against full 3D computational fluid dynamics (CFD) simulations. This work concludes with the combination of new Couette flow force approximations and similar results for the known Poiseuille flow (i.e. fluid flow induced by a pressure difference) to derive new relations for a combined Couette-Poiseuille flow force. This article is addressed to natural scientists and engineers that are interested in the application of conformal mappings and Taylor-expansions with the help of `SymPy` when solving partial differential equations analytically.

Index Terms—Physical modeling, Stokes equation, Eccentric annulus, Flow force, Conformal mapping, SymPy

Introduction

In times of digitization and wide use of numerical methods in physics, the question emerges whether analytical tools, such as Taylor-expansions or conformal mappings, are still of interest and how they can be utilized in industrial and academic research.

Computational power has increased significantly in the last years and many physical problems, ranging from electromagnetism to fluid dynamics and structural mechanics, can be solved directly by numerically integrating the corresponding three-dimensional PDEs, i.e. the Maxwell, Navier-Stokes or elasticity equations. However, when modeling physical systems such as hydraulic valves, transmission systems, engines, cars or planes, a direct 3D-approach for all relevant physical effects is still too difficult. In these situations, look-up tables containing a limited set of 3D-results are often included into 1D-system models for later interpolation. Alternatively, analytical approximations are used which are already included in the corresponding system simulation tools (e.g. Simcenter Amesim, [K19], [LGK21]).

Figure 1 schematically shows such a valve system with input data from look-up tables and 1D-component symbols.

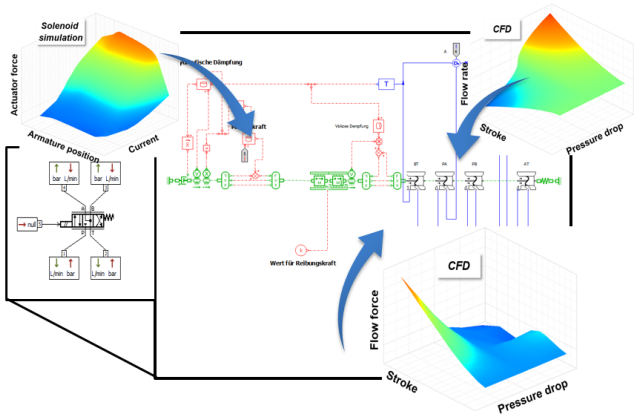

Fig. 1: *Valve system model with input data from look-up tables and 1D-component symbols as used in standard system simulation tools*

Hence, in modern system modeling there are currently two main applications of analytical approximations:

- Analytical approximations are included in the system simulation software components themselves, or
- The user includes look-up tables for interpolation, entirely or partially generated with analytical approximations.

In this work we will focus on analytical approximations of flow forces that act upon the inner cylinder in an eccentric annular flow domain. Such forces are of interest in, for example, hydraulic valves that are electromagnetically actuated; see Figure 2. When the armature moves within the oil-filled interior of the polecap, that movement causes a Couette-type annular flow, i.e. a viscous flow due to motion of a solid body, between both components.

For an analytical treatment, this geometry has to be simplified considerably. Both armature and polecap are therefore modeled as solid and hollow cylinders respectively. Since in realistic scenarios, perfect concentricity between these two parts is rarely obtained, the armature can be supported eccentrically within the poletube. A cross-sectional cut perpendicular to the symmetry axes of both cylinders then leads to Figure 3.

* *Corresponding author: zoufine.lauer-bare@hilite.com*
‡ *Hilite International, Weberstrasse 17, Nürtingen, Germany*

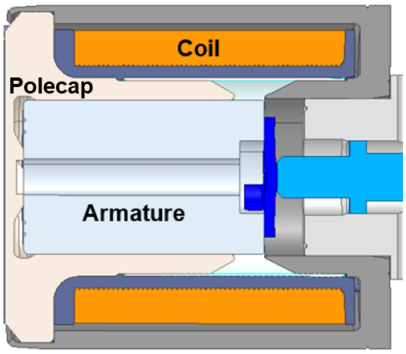

Fig. 2: *Armature and polecap in an electromagnetic actuator; the interior of the polecap is filled with oil (not shown here)*

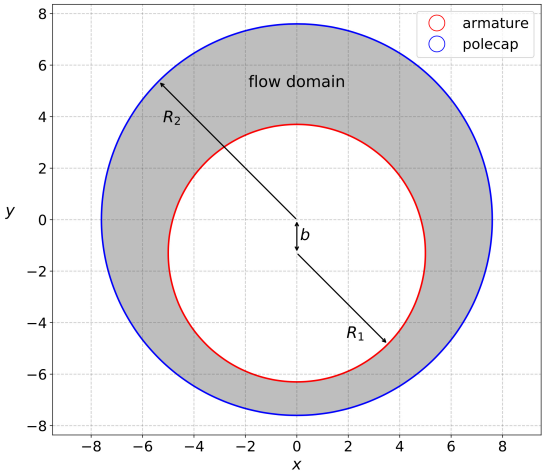

Fig. 3: *Cross-section of the simplified geometry of Figure 2, leading to an eccentric annular flow domain*

It shows the general case where an inner cylinder of radius R_1 is vertically displaced by a distance b from the center of an outer cylinder with radius R_2. The eccentric annular flow domain is contained in the region between these two cylinders.

In leading system simulation tools, the flow force that acts upon the inner cylinder in Figure 3 is typically approximated by the relation

$$F_{system} = -2\pi \frac{R_1 l \mu u_R}{\delta} . \tag{1}$$

Here μ denotes the viscosity of the fluid, l the common length of both cylinders, u_R the velocity and $\delta = R_2 - R_1$ the annular gap, i.e. the difference between outer and inner radius. Utilizing the capabilities of the open-source Computer Algebra System `SymPy` (as done e.g. in [MSP17]), we answer the following two questions:

1) How is Equation (1) related to the corresponding Stokes equation?
2) Does eccentricity $\varepsilon = b/\delta$ change this dependency and, if so, how exactly?

Furthermore, the velocities and forces obtained by solving the Stokes problem (i.e. the linear part of the Navier-Stokes system) with `SymPy` are compared to corresponding numerical solutions of the full, nonlinear Navier-Stokes equations, obtained from the commercially available Finite Volume tool ANSYS-CFX. Finally this article concludes with a note on the eccentric annular Poiseuille flow (that is a flow due to a pressure difference) and finishes with a comment on combined Couette-Poiseuille flow velocities and forces.

Material and methods

In order to solve the Stokes problem

$$\begin{aligned} -\mu\Delta u &= \frac{dp}{l} && \text{for } R_1 < \sqrt{x^2+(y+b)^2} \text{ and } \sqrt{x^2+y^2} < R_2 \\ u &= 0 && \text{for } \sqrt{x^2+y^2} = R_2 \\ u &= u_R && \text{for } \sqrt{x^2+(y+b)^2} = R_1 , \end{aligned} \tag{2}$$

the following `SymPy` functions and libraries were used: `im`, `re`, `subs`, `simplify` and `lambdify`. For the postprocessing the `SymPy` functions `diff` and `series` were particularly useful. Additionally, the `latex` function allowed to use the latex code of the formulae. For the interactive development with `SymPy` the `Jupyter Notebook` is used as GUI; there the `latex` math rendering proved to be very useful. The visualization is done with `NumPy` [HMW20] and `Matplotlib` [H07]. Code snippets are provided within the text in the subsequent sections. In addition, supplemental Python examples are available at this public GitHub repository[1].

The theoretical methods used here are conformal mappings (inspired by [PHW33] and [BC09]) and Taylor-expansions, following [LGK21]. Equations (2) describe *Couette flow* when $dp = 0$ and $u_R \neq 0$ and *Poiseuille flow*, when $dp \neq 0$ and $u_R = 0$. Furthermore, Equations (2) describe *Couette-Poiseuille flow* when $dp \neq 0$ and $u_R \neq 0$.

Solution of the Stokes problem within a concentric annulus for Couette-type flow

The solution of the Stokes problem within a concentric annulus for a Couette-type flow is well known, e.g. [LL87], and given by

$$u(r) = u_R \frac{\ln(r/R_2)}{\ln(R_1/R_2)} , \tag{3}$$

where $r = \sqrt{x^2+y^2}$. This can easily be checked by using the `diff` function of `SymPy`. Keep in mind, that the natural logarithm in Equation (3) is denoted by `log` there.

```
import sympy as sym
u_R, R1, R2, x, y = sym.symbols('u_r, R1, R2, x,
                                 y', real=True)
u = u_R * sym.log(sym.sqrt(x**2 + y**2)/R2)
        / sym.log(R1/R2)
laplacian = sym.diff(u, x, 2) + sym.diff(u, y, 2)
```

It then follows that

```
>>> sym.simplify(laplacian)
```

$$0$$

as expected. Further analytical solutions to the Laplace problem for other simple domains such as circles or rectangles can be found in e.g. [G13], [CB81] or [PP12].

1. https://github.com/zolabar/ConformalMappingSympy

Transformation of the eccentric annulus to a simple domain with conformal mappings

In the following two Sections we will show with SymPy how the Couette flow problem within an eccentric annular domain can be transformed into a problem within a concentric annular region or within a rectangle. In these simple geometries analytical solutions to this problem are well-known. In order to transform the domains we make use of complex analysis, inspired by the French mathematician Jacques Hadamard (1865-1963):

> *The shortest path between two truths in the real domain passes through the complex domain.*

The ideas and strategies of conformal mappings using SymPy are mostly described in the following Section, where a Möbius transform is used.

Transformation to a concentric annulus with Möbius transforms

Using a Möbius transform (also called a bilinear transformation) in the form of

$$w(z) = \xi + \mathrm{i}\,\eta = \frac{z + \mathrm{i}\,a}{a z + \mathrm{i}} \quad (\text{with } z = x + \mathrm{i}\,y)\,, \tag{4}$$

an eccentric annulus in the complex z-plane can be mapped onto a concentric annulus in the corresponding w-plane. The Möbius transform used here is a slightly adapted version of the one presented in [BC09]; a is a constant (given in [BC09]) and will be defined further down in this Section.

First of all, we will need some additional symbols for working with complex numbers and for the constant a.

```
z, a = sym.symbols('z, a', real=True)
```

Scaling the geometry in such a way that the outer circle ends up having a radius of 1

```
w = (z + sym.I * a)/(a * z + sym.I)
w = w.subs(z, x/R2 + sym.I * y/R2)
```

and separating real and imaginary part with SymPy functions

```
xi_ = sym.simplify(re(w))
eta_ = sym.simplify(im(w))
```

one arrives at

$$\xi = \frac{a x^2 + (R_2 + a y)\,(R_2 a + y)}{a^2 x^2 + (R_2 + a y)^2} \tag{5}$$

$$\eta = \frac{x\,(-R_2 - a y + a\,(R_2 a + y))}{a^2 x^2 + (R_2 + a y)^2}\,. \tag{6}$$

The latex rendering in the Jupyter Notebook shows directly the result of code in proper mathematical symbols, for instance

```
>>> sym.simplify(im(w))
```

$$\frac{x\,(-R_2 - a y + a\,(R_2 a + y))}{a^2 x^2 + (R_2 + a y)^2}$$

After the scaling, the Möbius transform constant a reads as

$$a = \frac{R_2 \left(\sqrt{\left(1 - \left(-\frac{R_1}{R_2} + \frac{b}{R_2}\right)^2\right)\left(1 - \left(\frac{R_1}{R_2} + \frac{b}{R_2}\right)^2\right)} + c_M \right)}{2b} \tag{7}$$

with c_M given by

$$c_M = \left(-\frac{R_1}{R_2} + \frac{b}{R_2}\right)\left(\frac{R_1}{R_2} + \frac{b}{R_2}\right) + 1\,. \tag{8}$$

Applying the Möbius transformation (4) to the boundaries leads to a concentric annular flow domain in the w-plane with inner radius 1 and outer radius R, given by

$$R = \frac{R_2 \left(\sqrt{\left(1 - \left(-\frac{R_1}{R_2} + \frac{b}{R_2}\right)^2\right)\left(1 - \left(\frac{R_1}{R_2} + \frac{b}{R_2}\right)^2\right)} - c_M \right)}{2R_1}\,. \tag{9}$$

This new flow domain is depicted in Figure 4.

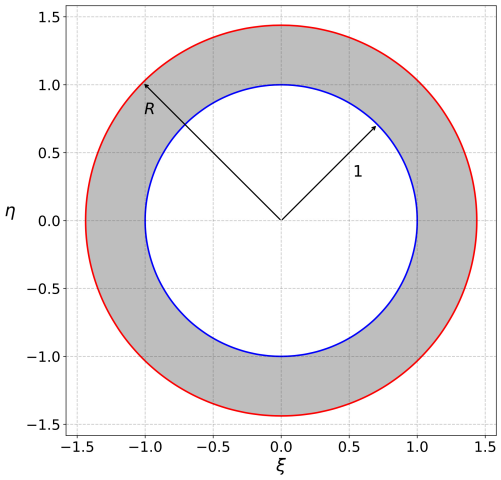

Fig. 4: *Concentric annular flow domain after Möbius transformation; keep in mind that armature and polecap are swapped in the w-plane*

Conformal mappings preserve harmonic functions, so the Stokes equation in the w-plane is of the same form as in the z-plane. However, Equation (4) interchanges inner and outer boundaries. This will affect the corresponding boundary conditions one needs to specify there so that the Stokes-problem in the w-plane is given by

$$\begin{aligned} -\Delta u &= 0 \quad && \text{for } 1 < \rho < R \\ u &= 0 \quad && \text{for } \rho = 1 \\ u &= u_R \quad && \text{for } \rho = R\,. \end{aligned} \tag{10}$$

Using the structure of Equation (3), the velocity in the w-plane is given by

$$u(\rho) = u_R \frac{\ln(\rho)}{\ln(R)}\,, \tag{11}$$

where $\rho = \sqrt{\xi^2 + \eta^2}$.

With the parameters specified in Table 1, the velocity in the w-plane (i.e. Equation (11)) can be used as an example for visualization and further evaluation.

The very convenient SymPy function lambdify is used to compute numerical values that are postprocessed by Matplotlib and depicted in Figure 5. The term R_ in the following code block denotes the numerical expression of the outer radius in the w-plane (see Equation (9)).

```
xi, eta = sym.symbols(xi, eta, u_R, real=True)
u_w = u_R * sym.log(sym.sqrt(xi**2 + eta**2))
          / sym.log(R)
```

Parameter	Value	Unit
R_1	5	mm
R_2	7.6	mm
b	1.3	mm
u_R	0.4	m/s

TABLE 1: *Geometry parametrization and imposed velocity for the simulations presented in this Section*

```
u_w = u_w.subs(u_R, 0.4).subs(R, R_)
u_w = sym.lambdify((xi, eta), u)
```

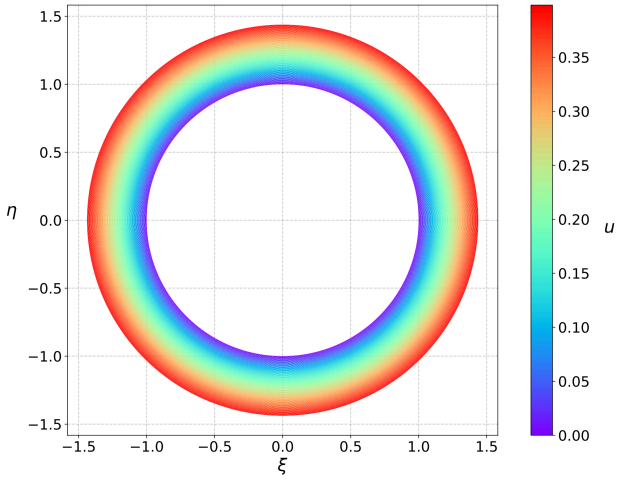

Fig. 5: *Flow velocity in concentric annulus (w-plane); the boundary condition ($u_R = 0.4\ m/s$) is applied to the outer cylinder, see Equation (10)*

At this stage it is pointed out that when working symbolically with `SymPy` one has to separate consistently between *expressions* and *symbols*. For instance `xi` and `eta` are symbols whereas `xi_` and `eta_` are expressions. The user can replace symbols by corresponding expressions when it best suits him/her. To avoid confusion, in this work the associated expression to a symbol `s` is tagged with an underline `s_`.

Now simply expressing ξ, η in (11) in terms of x and y (see Equation (5)), one easily obtains the fluid velocity in the eccentric annulus.

```
u = u_w.subs(xi, xi_).subs(eta, eta_)
u = sym.lambdify((x, y), u)
```

Figure 6 depicts the velocity distribution in the original z-plane. As one can see, the fluid gets dragged along the inner cylinder with the prescribed speed of 0.4 m/s. The velocity distribution then continuously drops down when moving radially outwards until it reaches zero along the outer cylinder.

Mapping rectangles onto eccentric annuli by bipolar coordinate transformations

Another way of solving this problem utilizes conformal mappings related to bipolar coordinates. These coordinates are described in [PHW33] and are commonly used in elasticity theory (e.g. [L13] and [TG10]). For this contribution, we slighty adapted this

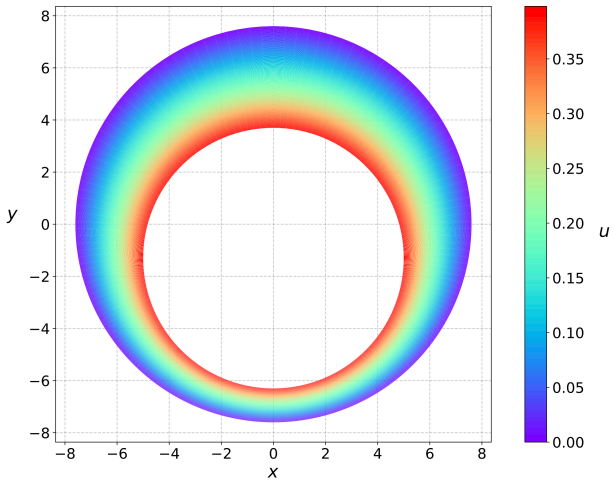

Fig. 6: *Flow velocity in eccentric annulus (z-plane); here the fluid moves with $u_R = 0.4\ m/s$ along the inner cylinder, as required by Equation (2)*

transformation in such a way that it can be applied to the eccentric annulus of Figure 3. The mapping is given by

$$z = c \cdot \tan\left(\frac{w}{2}\right) - \mathrm{i}\gamma \quad (\text{with } w = \xi + \mathrm{i}\eta), \tag{12}$$

where γ, c are constants from [PHW33] which are explicitly given in [W06] and [SL78]; the term $\mathrm{i}\gamma$ is added by the authors. Using this transformation, a properly chosen rectangular domain gets mapped onto an eccentric annulus; see Figure 7 for the domain in the w-plane. The boundaries are color-coded in order to visualize how the mapped borders are traversed in the z-plane. In addition the vertices are labelled and some coordinate lines are highlighted as well.

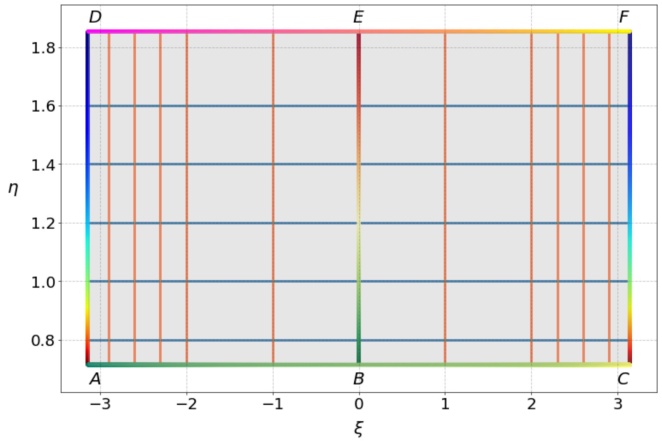

Fig. 7: *Rectangular domain in w-plane with color-coded boundaries, labelled vertices and some coordinate lines*

This domain gets transformed as shown in Figure 8. The vertices A and C (as well as D and F) are mapped onto the same respective points, i.e. $A' = C'$ and $D' = F'$. The color-coding shows that inner and outer cylinder are traversed counter-clockwise when moving in positive ξ-direction in the w-plane.

Furthermore the left and right vertical boundaries in the w-plane are identified in the z-plane, so periodic boundary conditions need to be applied to any PDE one wants to solve on the simple rectangle.

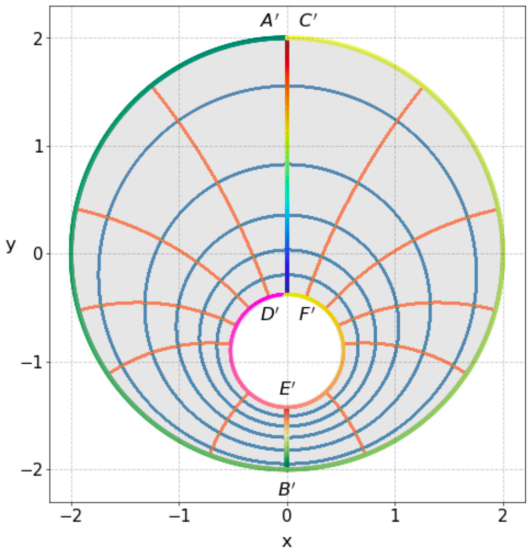

Fig. 8: *Mapped boundaries and coordinate lines in z-plane; the color-coding visualizes how the mapped borders are traversed here*

Please note that for demonstrational purposes the radius of the inner circle in Figure 8 is reduced in order to indicate how the coordinate lines are distorted. For conformal mappings however, although distances between corresponding points and lengths of curves are changing, the intersecting angle between any two curves is preserved.

Further details on the relation between conformal mappings and bipolar coordinates can be found in e.g. [CTL09]. Inverting Equation (12) and separating real and imaginary parts as in the previous Section one gets

$$\xi = -\arctan_2\left(2cx, c^2-x^2-(\gamma+y)^2\right) \tag{13}$$

$$\eta = \frac{1}{2}\ln\left(\frac{x^2+(y+\gamma+c)^2}{x^2+(y+\gamma-c)^2}\right). \tag{14}$$

Here, $\arctan_2(y,x)$ is the 2-argument arctangent which returns the polar angle of a point with Cartesian coordinates (x,y).

The constants from [W06] and [SL78] read as

$$F = \frac{1}{2b}\left({R_2}^2-{R_1}^2+b^2\right) \tag{15}$$

$$c = \sqrt{F^2-{R_2}^2} \tag{16}$$

$$\alpha = \frac{1}{2}\ln\left(\frac{F+c}{F-c}\right) \tag{17}$$

$$\beta = \frac{1}{2}\ln\left(\frac{F-b+c}{F-b-c}\right) \tag{18}$$

$$\gamma = c\coth(\alpha). \tag{19}$$

In the w-plane the corresponding Stokes-problem within the rectangular domain of Figure 7 is then prescribed by

$$\begin{aligned} -\mu\Delta u &= 0 \quad \text{for } \xi,\eta\in[-\pi,\pi]\times[\alpha,\beta] \\ u &= 0 \quad \text{for } \eta=\alpha \\ u &= u_R \quad \text{for } \eta=\beta \\ u(-\pi,\eta) &= u(\pi,\eta) \\ \frac{\partial u(-\pi,\eta)}{\partial\xi} &= \frac{\partial u(\pi,\eta)}{\partial\xi}. \end{aligned} \tag{20}$$

The last two equations specify the periodic boundary conditions one has to supply additionally. The solution to the system of equations (20) is easily obtained and given by the simple relation

$$u(\xi,\eta) = \frac{u_R(\eta-\alpha)}{\beta-\alpha}. \tag{21}$$

Figure 9 shows a `Matplotlib`-visualization of the velocity distribution in the w-plane which is constant along ξ and increases linearly with η.

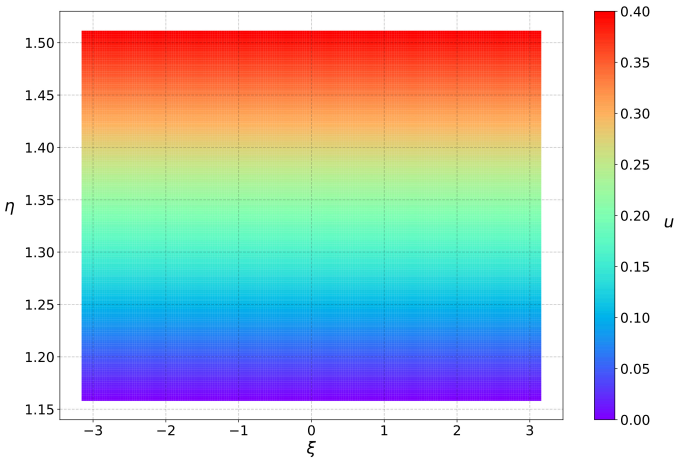

Fig. 9: *Flow velocity in rectangular domain (w-plane); here the proper boundary condition $u_R = 0.4\ m/s$ is applied to the upper boundary*

By again expressing η in terms of x and y, one obtains the very same velocity distribution in the eccentric annulus (in the z-plane) as already depicted in Figure 6.

It is interesting to remark, that Equations (11) and (21) look somehow related to each other due to the logarithm in both relations. However it is not immediately evident that they are actually identical. Nevertheless, due to existence and uniqueness theorems for the Stokes equation from [L14], one knows that relations (11) and (21) are in fact the same.

Figure 10 compares these two analytically obtained velocities with results from a 3D computational fluid dynamics simulation (using ANSYS CFX) solving the full Navier-Stokes system. For these computations a velocity of $u_R = -0.4$ m/s is prescribed onto the inner cylinder as boundary condition. All obtained velocities are evaluated along the symmetry axis of the annulus across the larger gap. The inner boundary is then reached on the left side, the outer boundary is hit on the right side of this Figure.

As one can see, the two analytical approaches lead to the same velocity distribution across the larger gap and both boundary conditions are met exactly. On the other hand, due to the finite mesh size particularly at the outer radius R_2, the boundary condition there is only approximately satisfied.

In the next Section, the corresponding flow force is obtained with `SymPy`-driven postprocessing and then compared again to the forces obtained by 3D-CFD and numerical evaluation.

Postprocessing

Force calculation and comparison with 3D-CFD

The relation for the annular flow force that acts upon the armature in Figure 4 is well known ([PHW33] or a more recent work

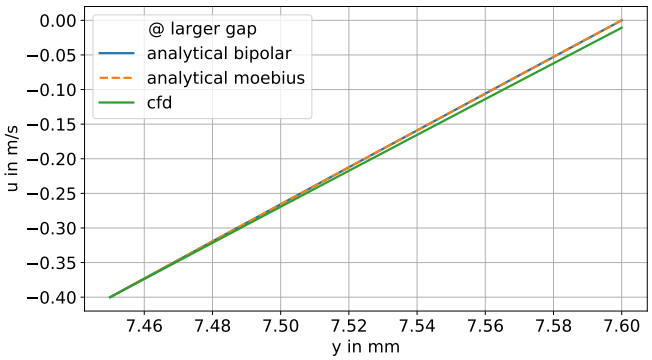

Fig. 10: *Flow velocity across the large gap within an eccentric annulus (eccentricity $\varepsilon = 0.5$); armature on the left, polecap on the right*

[LGK21]) and is given by

$$F_e = -\int_0^l \int_0^{2\pi} \left(\mu \rho \frac{\mathrm{d}}{\mathrm{d}\rho} u(\rho) \right)_{\rho=R} \mathrm{d}\varphi \, \mathrm{d}z. \tag{22}$$

This equation can be implemented in `SymPy` using the velocity distribution from Equation (11).

```
>>> u_w = u_R * sym.log(rho)/sym.log(R)
>>> u_w
```

$$u_R \frac{\ln(\rho)}{\ln(R)}$$

Using the `diff`, `subs` and `integrate` functions from `SymPy` then leads to

```
>>> Fe = mu * sym.diff(u_w, rho)
>>> Fe = (rho * Fe).subs(rho, R)
>>> Fe = sym.integrate(Fe, (z, 0, l))
>>> Fe = -sym.integrate(Fe, (phi, 0, 2 * pi))
>>> Fe
```

$$-2\pi \frac{l \mu u_R}{\ln(R)}$$

Substituting the relation for R into F_e, the flow force of the eccentric annular Couette flow is obtained. It can be manually adapated to the esthetic preferences of the authors, e.g.

$$\begin{aligned} F_{Couette} &= -2\pi \frac{l \mu u_R}{\ln[(c_f R_2)/R_1]} \\ c_f &= -\frac{1}{2}\left(c_1 c_2 + \sqrt{\left(1-c_1^2\right)\left(1-c_2^2\right)} + 1 \right) \\ c_1 &= \frac{R_1}{R_2} + \frac{b}{R_2} \\ c_2 &= -\frac{R_1}{R_2} + \frac{b}{R_2}. \end{aligned} \tag{23}$$

Equation (23) therefore answers the second question posed in the Introduction: *The flow force is decisively influenced by the eccentricity*.

Alternatively, the Couette flow force can be derived from Equation (21), which is obtained from solving the equivalent Stokes-problem in bipolar coordinates and for this case it is given by

$$F_{Couette2} = -2\pi \frac{l \mu u_R}{\beta - \alpha}. \tag{24}$$

Parameter	Value	Unit
μ	11.53	mPa · s
l	11.95	mm
ρ	807	kg/m^3

TABLE 2: *Additional fluid- and geometry-parameters used for the 3D-CFD simulations*

With the data in Table 1 and Table 2, Figure 11 shows a comparison between the analytically obtained relations (23) and (24) and results from 3D-CFD simulations of the full Navier-Stokes system for a wide range of different eccentricities.

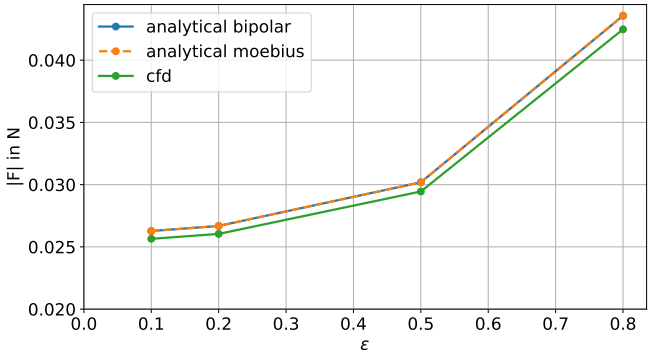

Fig. 11: *Flow force according to Equation (22), acting on the inner cylinder of an annulus with varying eccentricity ε*

Again, both analytical relations agree perfectly but since the numerical CFD-results for the velocity slightly diverge from the analytical solution especially towards the outer boundary (as seen in Figure 10), the flow force computed from this data also shows smaller deviations.

Taylor-expansions and small gaps

Equation (23) is even defined for the concentric case. Substituting $b = 0$ into this relation and simplifying the resulting expression leads to

$$F_c = -u_R \frac{2\pi \mu l}{\ln(R_2/R_1)}. \tag{25}$$

In order to finally answer the first question of the Introduction, i.e. how Equation (1) is related to the Stokes equation, the `series` function of `SymPy` is used. With `series`, a Taylor-expansion of F_c in $\delta = R_2 - R_1$ around $\delta = 0$ can be performed

```
>>> sym.series(Fc.subs(R2, R1 + delta), delta, 0, 2)
```

$$\frac{\pi \delta l \mu u_R}{6R_1} - \pi l \mu u_R - \frac{2\pi R_1 l \mu u_R}{\delta} + O\left(\delta^2\right) \tag{26}$$

The answer to the aforementioned question then is: (1) *is the leading term of a Taylor-expansion of the concentric annular Couette flow force around $\delta = 0$.*

The contribution of this article closes with some additional remarks on eccentric annular Poiseuille flow and new possibilities of combining the results of the last Sections with results from [PHW33] and [LGK21].

Additional remarks on Poiseuille flow

Eccentric annular Poiseuille flow velocity

In various circumstances Couette flow may also induce a secondary flow driven by a pressure difference; a so-called *Poiseuille flow*. This particular type is of interest in many areas and we'll briefly show how the corresponding solution presented in [PHW33] is derived conceptually as well as how it can be implemented with the help of `SymPy`.

As far as we know, most of the current literature either refers to the aforementioned paper only by using its derived results (e.g. the volume flow relation found in [W06]) or by solving the Poiseuille problem numerically (as done in [TKM19]). The fact, that in the current context blood coagulation and hemodynamics are omnipresent in the media, eccentric annular blood flow in arteries is extensively studied ([TKM19]) and flow forces that act upon the arteries are of great medical interest (e.g. [S11]), makes it even more interesting to retrace the existing formulae of [PHW33], which are tedious to use when implemented by hand.

In the case of Poiseuille flow, the righthand-side of the corresponding Stokes equation is non-homogeneous ($dp \neq 0; u_R = 0$); see also Equation (2). Hence, we need to deal with a different mathematical problem here compared to the previous Sections.

However, it possible to reduce the Poiseuille problem to an equivalent Couette problem with prescribed velocities on the boundaries (e.g. [M96]). That is the idea followed by [PHW33], who seek a solution of the form

$$u = \Psi - \frac{dp}{4\mu l}(x^2+y^2). \tag{27}$$

Here, Ψ is a harmonic function in the w-plane found by solving Laplace's equation in ξ and η. By using the conformal mapping of Equation (12) an appropriately chosen rectangle in the w-plane gets mapped onto an eccentric annulus in the z-plane, thereby preserving the harmonicity of Ψ.

It then follows that $\Delta u = dp/(\mu l)$ in the z-plane and the boundary conditions for Ψ result from the task of eliminating the auxiliary term $-\frac{dp}{4\mu l}(x^2+y^2)$ on the boundaries associated with inner and outer radius.

For further evaluation, Ψ is decomposed by [PHW33] into a sum of three harmonic functions

$$\Psi = 4\cdot\Psi_1 + 4\cdot A\cdot\eta + 4\cdot B. \tag{28}$$

Using this particular form of Ψ, the final relation for the Poisseuille-flow velocity derived in [PHW33] can be symbolically expressed via

```
xi, eta, b = sym.symbols('xi, eta, b', real=True)
A, B, C = sym.symbols('A, B, C', real=True)
alpha, beta, c = sym.symbols('alpha, beta, c',
                             real=True)
Psi_1, mu, l, dp = sym.symbols('Psi_1, mu, l, dp',
                               real=True)
k, m, n = sym.symbols('k m n', integer=True)


>>> u = Psi_1 + A * eta + B
>>> u = u - (sym.cosh(eta) - sym.cos(xi))
          / (4 * (sym.cosh(eta) + sym.cos(xi)))
>>> u = (dp/(mu * l)) * c**2 * u
>>> u
```

$$\frac{c^2 dp\left(A\eta + B + \Psi_1 - \frac{-\cos(\xi)+\cosh(\eta)}{4\cos(\xi)+4\cosh(\eta)}\right)}{l\mu} \tag{29}$$

Afterwards the expressions for the three separate components A, B and Ψ_1 can finally be substituted into (29). In the following code the `SymPy` function `Sum` is used, which simplifies the implementation of Fourier-type series in analytical formulae significantly.

```
s1, s2 = sym.symbols('s1, s2', real=True)
Psi_1_ = sym.cos(n * xi)
         / (sym.sinh(n * (beta - alpha))) * (s1 + s2)
Psi_1_ = sym.Sum((-1)**n * (Psi_1_), (n, 1, m))


>>> Psi_1_
```

$$\sum_{n=1}^{m} \frac{(-1)^n (s_1+s_2)\cos(n\xi)}{\sinh(n(\beta-\alpha))} \tag{30}$$

with

$$\begin{aligned} s_1 &= e^{-\beta n}\sinh(n(\eta-\alpha))\coth(\beta) \\ s_2 &= -e^{-\alpha n}\sinh(n(\eta-\beta))\coth(\alpha). \end{aligned}$$

The constants from [W06], [SL78] and [PHW33] read as

$$\begin{aligned} A &= \frac{\coth(\alpha)-\coth(\beta)}{2\alpha-2\beta} \\ B &= \frac{-\alpha(1-2\coth(\beta))+\beta(1-2\coth(\alpha))}{4\alpha-4\beta}. \end{aligned}$$

Adding the various pieces together, an example of Piercy's Poiseuille flow velocity (Equation (27)) in the w-plane is depicted in Figure 12.

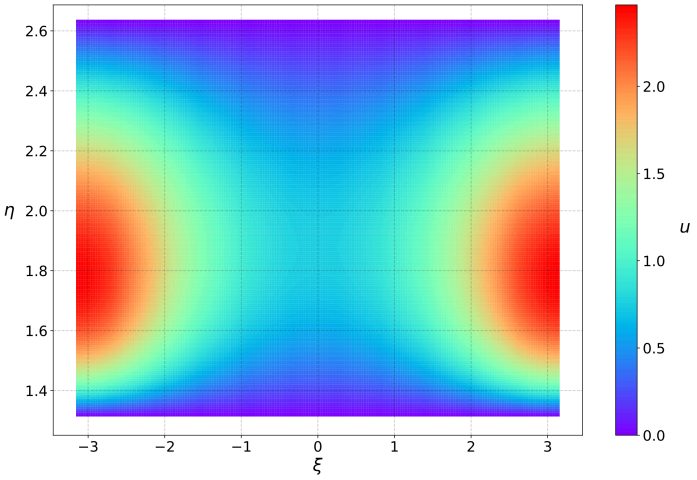

Fig. 12: *Flow velocity for the Poiseuille problem in rectangular domain (w-plane); it vanishes on upper and lower boundary and is periodic in ξ*

And last but not least, again expressing ξ, η in x and y, the velocity distribution in the eccentric annulus (i.e. in the z-plane) together with some isocontours is shown in Figure 13.

The method described here is not only restricted to fluid dynamics. In elasticity theory, which inspired the work of [PHW33], Ψ is the harmonic conjugate of the so-called warping- or St. Venant torsion-function ϕ (see [L13] or [M77]), specified by

$$\frac{\partial\Psi}{\partial y} = \frac{\partial\phi}{\partial x} \quad \text{and} \quad \frac{\partial\Psi}{\partial x} = -\frac{\partial\phi}{\partial y}.$$

The warping function helps to describe the elongation of an elastic cylinder that is also twisted. A practical implementation of ϕ can be found in e.g. [B14] and [BPO16] where it is called $n_{1,4}^{inner}$ and where analytical approximations are compared to results from 3D-simulations obtained with COMSOL.

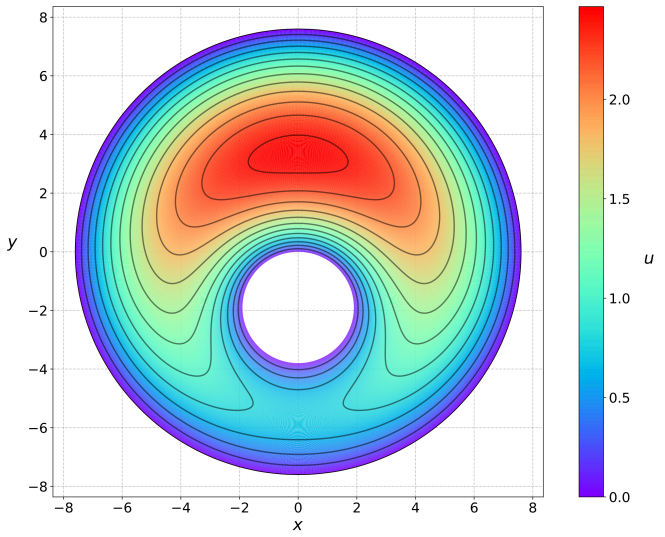

Fig. 13: *Flow velocity and isocontours for the Poiseuille problem in eccentric annulus (z-plane); most of the fluid flow occurs through the large gap*

Eccentric Couette-Poiseuille flow: Superposition

The velocity for eccentric Couette-Poiseuille flow can easily be found by superposing Equation (29) with one of the two Couette flow velocities derived in this contribution by utilizing `SymPy`.

The following relation

$$u_{Coue-Pois} = \frac{c^2 dp \left(\Psi - \frac{-\cos(\xi)+\cosh(\eta)}{\cos(\xi)+\cosh(\eta)}\right)}{4l\mu} + \frac{u_d\,(\eta-\alpha)}{\beta-\alpha} \tag{31}$$

shows such a superposed Couette-Poiseuille flow velocity, where both velocities where obtained by using the bipolar coordinate transformation (12) that maps rectangles onto eccentric annuli.

Combining Equation (24) with the flow force from [PHW33], the overall exact analytical eccentric annular Couette-Poiseuille flow force that acts upon the inner cylinder is given by

$$F_{Coue-Pois} = F_{piercy} - \frac{2\pi l\mu u_R}{\beta-\alpha} \tag{32}$$

where

$$F_{Piercy} = -\pi\Delta p\left({R_1}^2 - \frac{b\cdot c}{\beta-\alpha}\right). \tag{33}$$

Since the conformal mapping (12) is not defined for the concentric case $b=0$, this drawback also translates to the corresponding forces in Equations (32) and (33). The relation above therefore is only defined for eccentric cases.

However, the Couette flow force obtained with the Möbius transform, i.e. Equation (23), is defined for the concentric case as well. But since, to our knowledge, no one has ever constructed the Poiseuille flow velocity using a Möbius transform, the equivalent flow force (most likely defined for $b=0$ too) is not available.

Therefore, the best analytical approximation for the eccentric Couette-Poiseuille flow force, defined both for the eccentric and concentric case, that we can present here, is a combination of Equation (23) and a Taylor-expansion of Equation (33) in the relative eccentricity $\varepsilon = b/(R_2-R_1)$ around $\varepsilon=0$.

$$F_{Coue-Pois} \approx F_{Couette} + F_c\left(1+a(\kappa)\,\varepsilon^2\right). \tag{34}$$

Here, F_c is the well known Poiseuille flow force that acts upon the inner cylinder in the concentric case (e.g. [BSL07]) and $a(\kappa)$ is a function of the ratio $\kappa = R_1/R_2$ given by

$$F_c = -\pi\Delta p\left({R_1}^2 - \frac{\left(R_2^2-R_1^2\right)}{2\ln\left(R_2/R_1\right)}\right) \tag{35}$$

$$a(\kappa) = -(1-\kappa)\frac{\left(1-\kappa^2\right)+\left(1+\kappa^2\right)\ln\kappa}{2\left(\kappa^2+\frac{\left(1-\kappa^2\right)}{2\ln\kappa}\right)(1+\kappa)\ln^2\kappa}. \tag{36}$$

The particular approximation for the eccentric flow force due to a pressure gradient, i.e. $F_{Piercy} \approx F_c\left(1+a(\kappa)\,\varepsilon^2\right)$, was obtained for the first time in [LGK21].

To conclude this Section it is remarked, that again the useful `SymPy` function `series` can help in figuring out how $a(\kappa)$ is approximated in the relevant practical case where $R_1 \approx R_2$.

As shown in [LGK21], $a(\kappa)$ can be expanded in a Taylor-series around $\kappa = 1$.

```
>>> sym.series(alpha, kappa, 1, 3)
```

$$-\frac{1}{6} - \frac{5(\kappa-1)^2}{36} + \frac{\kappa}{6} + O\left((\kappa-1)^3;\kappa\to 1\right)$$

Hence, for $\kappa \approx 1$

$$a(\kappa) \approx \frac{\kappa-1}{6}$$

and (34) reduces to

$$F_{Coue-Pois} \approx F_{Couette} + F_c\left(1+\frac{\kappa-1}{6}\,\varepsilon^2\right). \tag{37}$$

Conclusion

This article showed that classical tools from mathematical physics, such as conformal mappings and Taylor-expansions, are still relevant and indispensable in times of digitization and wide use of numerics.

As an example, `SymPy` was used as a tool for symbolic mathematics in order to demonstrate that a popular approximation of the eccentric annular Couette flow force in modern system simulation tools is actually the leading-order term in a Taylor-expansion of the corresponding concentric annular force.

This force is calculated as special case of the more general eccentric annular Couette flow by postprocessing the resulting velocity distribution. Here, the velocity profile is analytically obtained by solving the equivalent Stokes problem with the help of conformal mappings, i.e. holomorphic functions in the complex plane.

The utilization of analytical methods is not solely restricted to fluid dynamics. Another application of `SymPy` in the context of PDEs in general could be homogenization. There, asymptotic expansions are substituted into the PDE and limiting problems are obtained in an algorithmical way, so `SymPy` might prove to be a valuable supporting tool. A starting point could be the introductory example from [BP89], which is worked out and compared to a FEM-solution obtained by COMSOL in [B14]. Furthermore, due to similar equations in axisymmetric electromagnetic problems (e.g. [LL84]), corresponding usage of conformal mappings and Taylor-expansions with `SymPy` is certainly possible there.

The authors think, that these methods may not only be applicable to mathematical physics but could be helpful in other areas as well, e.g. for understanding neural networks. Already available work described in [H10] and [H12] points in that direction and `SymPy` might be of great help in such areas, too.

REFERENCES

[BP89] Bakhvalov NS, Panasenko G. *Homogenisation: averaging processes in periodic media: mathematical problems in the mechanics of composite materials*, Kluwer Academic Publisher; 1989, https://doi.org/10.1007/978-94-009-2247-1

[B14] Bare Contreras DZ. *Asymptotic Analysis for Linearized Contact Problems in Thin Beams*, Fraunhofer Verlag; 2014, ISBN 978-3-8396-0762-6

[BPO16] Bare Z, Orlik J, Panasenko G. *Non homogeneous Dirichlet conditions for an elastic beam: an asymptotic analysis*, Applicable Analysis, 2016, 2625-36, https://doi.org/10.1080/00036811.2015.1105960

[BSL07] Bird RB, Stewart WE, Lightfoot EN. *Transport phenomena*, John Wiley & Sons; 2007, ISBN 978-0-470-11539-8

[BC09] Brown JW, Churchill RV. *Complex variables and applications*, McGraw-Hill, NY; 2009, ISBN 978-0-0733-8317-0

[CTL09] Chen JT, Tsai MH, Liu CS. *Conformal mapping and bipolar coordinate for eccentric Laplace problems*, Computer Applications in Engineering Education, 2009, 314-22, https://doi.org/10.1002/cae.20208

[CB81] Churchill RV, Brown JW. *Fourier series and boundary value problems*, McGraw-Hill, NY; 1981, ISBN 978-0-0780-3597-5

[G13] Greenberg MD. *Foundations of applied mathematics*, Dover; 2013, ISBN 978-0-4864-9279-7

[HMW20] Harris CR, Millman KJ, van der Walt SJ et al. *Array programming with NumPy*, Nature 585, 2020, 357–362, https://doi.org/10.1038/s41586-020-2649-2

[H10] Hirose A. *Recent progress in applications of complex-valued neural networks*, International Conference on Artificial Intelligence and Soft Computing, 2010, https://doi.org/10.1007/978-3-642-13232-2_6

[H12] Hirose A. *Complex-valued neural networks*, Springer Science & Business Media; 2012, https://doi.org/10.1007/978-3-642-27632-3

[H07] Hunter JD. *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, 2007, vol. 9, no. 3, 90-95, https://doi.org/10.5281/zenodo.592536

[K19] Krebs J. *Optislang in functional development of hydraulic valves*, RDO Journal Issue 2, 2019

[L14] Ladyzhenskaya OA. *The mathematical theory of viscous incompressible flow*, Martino Publishing; 2014, ISBN 978-1-6142-7671-5

[LL84] Landau LD, Lifshitz EM. *Electrodynamics of continuous media*, Pergamon Press, NY; 1984, ISBN 978-0-08-030275-1

[LL87] Landau LD, Lifshitz EM. *Fluid Mechanics*, Pergamon Press, NY; 1987, https://doi.org/10.1016/C2013-0-03799-1

[LGK21] Lauer-Baré Z, Gaertig E, Krebs J, Arndt C, Sleziona C, Gensel A. *A note on leakage jet forces: Application in the modelling of digital twins of hydraulic valves*, International Journal of Fluid Power, 2021, Vol. 22 (1), 113–146, https://doi.org/10.13052/ijfp1439-9776.2214

[L13] Love AEH. *A treatise on the mathematical theory of elasticity*, Cambridge University Press; 2013, ISBN 978-1-1076-1809-1

[MSP17] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T. *SymPy: symbolic computing in Python*, PeerJ Computer Science; 2017, https://doi.org/10.7717/peerj-cs.103

[M96] Milne-Thomson LM. *Theoretical Hydrodynamics*, Courier Corporation; 1996, ISBN 978-0-4866-8970-8

[M77] Muskhelishvili NI. *Some basic problems of mathematical elasticity theory*, Springer Science & Business Media; 1977, https://doi.org/10.1007/978-94-017-3034-1

[PHW33] Piercy NAV, Hooper MS, Winny HF. *LIII. Viscous flow through pipes with cores*, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 1933, https://doi.org/10.1080/14786443309462212

[PP12] Pikulin VP, Pohozaev SI. *Equations in mathematical physics: a practical course*, Springer Science & Business Media; 2012, https://doi.org/10.1007/978-3-0348-8285-9

[S11] Secomb TW. *Hemodynamics*, Comprehensive Physiology, 2011 Jan 17;6(2):975-1003, https://doi.org/10.1002/cphy.c150038

[SL78] Shah RK, London AL. *Laminar flow forced convection in ducts*, Supplement 1 to Advances in Heat Transfer, Academic Press, NY; 1978, https://doi.org/10.1016/C2013-0-06152-X

[TG10] Timoshenko S, Goodier JN. *Theory of elasticity*, McGraw-Hill, NY; 2010, ISBN 978-0-0707-0122-9

[TKM19] Tithof J, Kelley DH, Mestre H, Nedergaard M, Thomas JH. *Hydraulic resistance of periarterial spaces in the brain*, Fluids and Barriers of the CNS, 16, 2019, https://doi.org/10.1186/s12987-019-0140-y

[W06] White FM. *Viscous fluid flow*, McGraw-Hill, NY; 2006, ISBN 978-0-0724-0231-5

Programmatically Identifying Cognitive Biases Present in Software Development

Amanda E. Kraft[‡†], Matthew Widjaja[‡†*], Trevor M. Sands[‡], Brad J. Galego[‡]

Abstract—Mitigating bias in AI-enabled systems is a topic of great concern within the research community. While efforts are underway to increase model interpretability and de-bias datasets, little attention has been given to identifying biases that are introduced by developers as part of the software engineering process. To address this, we began developing an approach to identify a subset of cognitive biases that may be present in development artifacts: anchoring bias, availability bias, confirmation bias, and hyperbolic discounting. We developed multiple natural language processing (NLP) models to identify and classify the presence of bias in text originating from software development artifacts.

Index Terms—cognitive bias, software engineering, natural language processing

Introduction

Artificial intelligence (AI) and machine learning (ML) -based systems are increasingly supporting decision-making, reasoning, and evaluation of dynamic environments in objective manners. As AI-enabled systems are finding increasing use across domains and industries, there is concern that the objectivity of such systems may be negatively impacted by biases introduced by the developers either in the design of the system or in the training data itself. While efforts are underway to make AI/ML systems more interpretable and debias datasets, little research is directed at human-centric cognitive biases that developers unintentionally introduce as a part of the software engineering (SE) process. As a result, ensuring unbiased and transparent algorithmic decision-making is a complex challenge and has wide-ranging implications for the future use of AI in society.

Cognitive biases are systematic deviations from rationality in judgment, reasoning, evaluation, or other cognitive processes. For the myriad of cognitive biases described in literature[1], approximately 40 have been investigated in the SE domain[2]. We selected four of the most commonly reported cognitive biases in software engineering:

- **Anchoring Bias**: Tendency to rely too heavily on pre-existing or first information found when making a quantitative judgment[2].
- **Availability Bias**: Tendency to overestimate the likelihood of events based on the ease of which examples come to mind[3].
- **Confirmation Bias**: Tendency to search for and focus on information that confirms one's preconception(s) while ignoring or rejecting sources that challenge it[4].
- **Hyperbolic Discounting**: Tendency to prefer immediate payoffs over larger rewards at a later point[2].

These biases may be influenced by self-generated factors (e.g., past development experience), or externally generated factors (e.g., system documentation)[5]. A tool to detect biases in software must be capable of assessing multiple sources of information about the system, including commit messages, comments, in-source docstrings, external technical documentation, and diagrams. This study takes the first steps toward this objective by identifying cognitive biases in software commit messages and comments from previously completed projects.

The remainder of this paper is organized into three sections: research methods, results and discussion, and conclusions and implications for future work in this space.

Research Methods

In this section we discuss how data was initially gathered and curated prior to annotation, how manual annotation was performed using Prodigy, the process for reviewing and finalizing the consensus for data labels, and finally, the approach for developing machine learning classifier models applied to the annotated data to determine whether bias is present in a given sample.

Data Curation

To address the lack of research identifying cognitive biases in software artifacts developed as part of a naturally occurring development process, we collated data from two internally developed codebases. The first project ("Project A") was selected to represent the whole software engineering process for AI/ML-enabled systems (i.e., data management to feature extraction to model training and evaluation). The second project ("Project B") is similar in structure to the first, but the software artifacts gathered include only the latter half of the development cycle (i.e., feature extraction to model training and evaluation). The content from both codebases were collated into datasets based on the source of the development artifacts: commit messages, in-source code comments, and documentation strings (docstrings). Given the time limitations for this effort, we prioritized annotation of commit

† *These authors contributed equally.*
‡ *Lockheed Martin Advanced Technology Laboratories*
∗ *Corresponding author: matthew.widjaja@lmco.com*

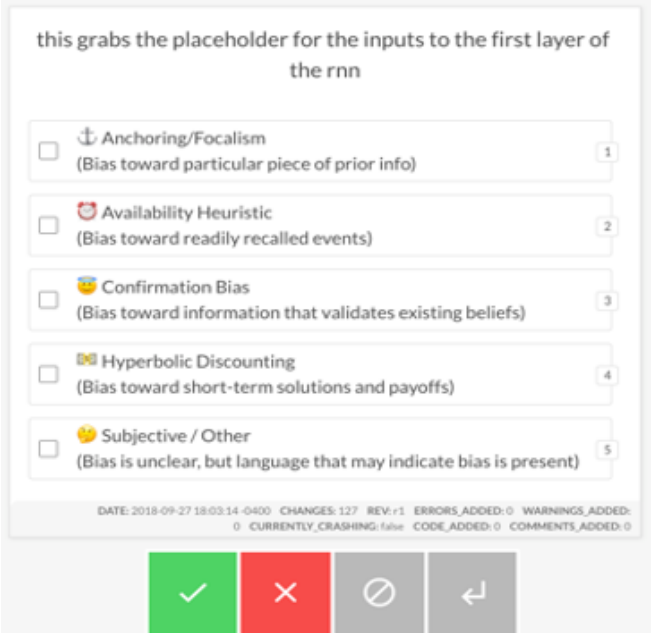

Fig. 1: *Example view of a comment in reviewer mode. The reviewer has three options: (1) accept via the green checkmark if bias is detected, (2) reject via the red X if no bias detected, and (3) ignore via the grey stop icon if the entry contains no intelligible content.*

messages for all datasets, while comments and docstrings were annotated for the second of the two internal projects.

Further, we identified an open-source dataset, Code Smell[6], to validate models trained on the content from the internal projects. This dataset contains commit messages extracted from refactoring activities across the software lifecycle for various open-source projects.

For all datasets, python scripts were developed to programmatically extract and format the text content from the artifacts. Specifically, the following operations were performed: commit message content had whitespace trimmed and artifact identifiers removed; comments spanning multiple lines were combined into a single entry; executable segments of code were removed; entries with non-ASCII characters were removed; references to individual names, collaboration teams, applications, and projects were redacted and replaced with an identifier string (e.g., “NAME1”).

Bias Annotation: Prodigy Setup

The processed text data was then annotated in Prodigy to produce a structured JSON document. Prodigy is a commercially licensed software tool for collaborative data annotation. A custom annotation recipe was developed to define the four biases described above as the possible labels; an additional label option, “Subjective/Other” was included to provide reviewers a chance to flag entries containing a form of bias other than the available options. Figure 1 provides an example of what individual reviewers see when annotating a given dataset using this custom recipe. For each entry, the reviewer must decide whether an entry is valid, and if so, if the language indicates that the author may have introduced bias into the system. When reviewers determined an entry contains bias, they selected one or more labels and pressed “accept”; otherwise, the reviewer pressed “reject” to indicate no language indicating bias was present.

Bias Annotation: Manual Annotation

A total of six reviewers were engaged in this project for the bias annotation process. All reviewers have at least two years of programming experience and are between the ages of 18-40. Two reviewers are female and four are male. Two reviewers are Asian, while the other four are White. Three of the six reviewers had some degree of involvement in developing the software for two of the internal projects discussed in this paper. Further, one of these reviewers was the software lead on Project B. To minimize personal biases when reviewing the development artifacts, all entries are anonymized and annotated in non-chronological order.

An annotation guide for classifying open-ended text entries was developed for reviewers to remain consistent. The guide provides examples of several biased commit messages such as:

- Anchoring Bias
  - "Extended module to allow a more traditional approach to interface engineering"
  - "Applying back-changes from my original fix patch"
  - "Correct the temperature unit - assumes anything under 45 is C"
- Availability Heuristic
  - "Renamed method to more sensible wording”
  - "Tighter coupling of variable names with other modules"
- Confirmation Bias
  - "The use of [X] rather than [Y] allows each module to reuse the same functionality without having to extend a base class"
  - "We’re now a bit smarter about the size of tables that we create by default, which was the root of the prior problems"
- Hyperbolic Discounting
  - "Throwing out the Key and Value classes for now to reduce the overall complexity"
  - "Modified function to account for type errors. Will likely have to recreate the db every time, unless other solutions come up"
  - "Module incorporated but fails"
  - "Quick and dirty method to add features"
- Subjective/Other
  - "I was too over-zealous with removing a module"
  - "Duplicate code is my nemesis..."

The guide reminds reviewers that they are to label if the language indicates the author may have introduced bias into the system, not if the language indicates the author may be addressing bias previously introduced. The guide further advises the reviewer to flag entries as invalid if they should be excluded from the training or testing datasets; the exclusion criteria include blank messages, machine-generated messages (e.g., automated branch merging messages), messages only containing an artifact or issue identifier, and “TODO” or “FIXME” comments with no accompanying description. Reviewers were also encouraged to accept samples that may be borderline cases, as a group consensus would decide final classification labels.

Bias Annotation: Finalizing Bias Labels

After all reviewers submitted their final annotations for a dataset, one reviewer was selected to finalize the labels to be used for training and testing models. For consistency, the same reviewer was selected to finalize labels on all datasets. The review process itself was facilitated by Prodigy, which offers a built-in review recipe, allowing a user to specify the annotation databases to use. With this recipe, Prodigy extracts all instances where an entry was marked as "accepted" or "ignored" by at least one reviewer. These are compiled and displayed similar to the initial review, noting which review session(s) indicated which label(s).

In the final review, a "best fit" label was selected, rather than accepting multiple labels for a single entry as allowed in the initial review stage. This decision was implemented in order to provide non-overlapping classification boundaries for model training and testing. The final reviewer followed a set of guidelines for determining best fit labels, such as cross-referencing the annotation guide or identifying the word or phrase that may have triggered the response when multiple reviewers selected different biases for a single message.

If the final reviewer thought the best fit label was ambiguous or if the label selected was only reported by themselves during the initial review process, the message was logged for additional review. These flagged messages were compiled in an Excel workbook along with the selected answer (first-degree label), the next best answer (second-degree label), and the labels marked by the initial review sessions. The workbook was sent to at least two individuals to respond to these entries, indicating their judgement of whether the first or second degree label was the best fit or if another label option may have been overlooked. Scoring of their responses was automated using the following rules: (1) if both agreed with the first degree label, it was kept; (2) if both agreed with the second degree label, the final label was switched; (3) if the first degree label was not "reject" and one accepted while the other rejected, the first degree label was kept. On the rare occasion when none of these conditions were met, the final reviewer decided the label selection based on the feedback.

The results of the final review (i.e., entries labeled as biased) were merged with the source dataset (i.e., non-biased entries) to comprise the training and testing datasets for modeling.

Models

To determine whether a tool can classify software artifacts as containing indicators of bias, we developed text classification models using spaCy. Binary and multi-class models were considered, where binary models were concerned with identifying the presence or absence of biased language and multi-class models concerned with identifying the type of bias present (if one is present at all). Anticipating that the class distributions would be highly imbalanced towards not containing bias, we implemented down sampling by taking the mean of the quantity of data present across each label type to improve model training. This method was randomized, with ten models trained on different data distributions.

Focusing on the ability of the trained models to perform on different codebases, we prioritized evaluating the models independently trained on the two internal commit datasets and applied each to the Code Smell dataset (i.e., as a test dataset). As a secondary task, we then combined the internal commits in a single training set and applied them to Code Smell. Additionally, to determine if commit messages can predict bias in comments,

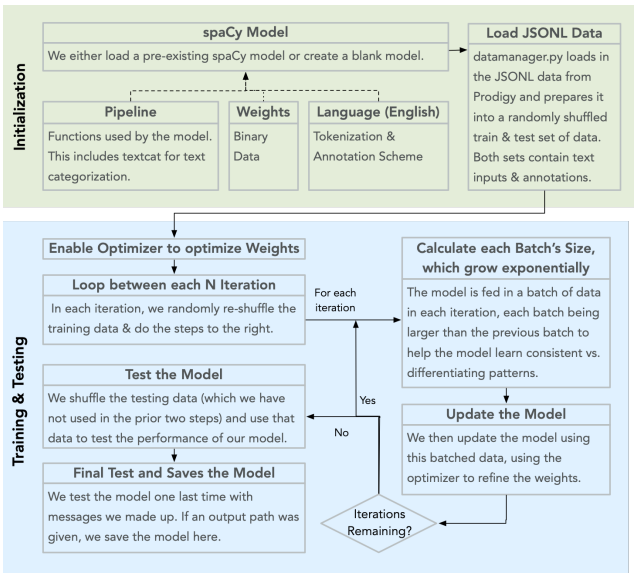

Fig. 2: *Overview of the spaCy NLP modeling workflow, broken up into initialization (green) and execution (blue) task phases.*

we trained a model on the internal commits and tested against comments for the same project. Finally, we evaluate the combined internal dataset against Code Smell.

We ran each model three times, each time using a different split of the dataset. This modeling process is illustrated in Figure 2. For each model, we report the mean F1 score and standard deviation across runs. We swept across three model hyperparameters during training:

1) **The maximum number of samples used to train**: This mitigates the impact of label imbalance, by limiting the total number of entries from each category before training the model. The considered caps included:
    1. The quantity of entries from all biases.
    2. The mean of the quantity of data from each category, including data which was not biased.
    3. The quantity of entries from the largest bias category.
    4. No capacity, use all data.
2) **Dropout**: This is the percentage of connections which are dropped from the neural network component of the ensemble learning and is used to prevent over-fitting. Typical sweep values are 20%, 40%, and 60%.
3) **The size of the training batches and their compounding rate at each epoch:** This determines how much data is passed to the trainer at each iteration from a minimum (batch start size) to a maximum (batch stop size) with a given rate of growth (compounding rate). For all models, the compounding rate was left at the spaCy recommended value of 1.001.

Results and Discussion

In this section we discuss the results of data annotation and the classifier models. Statistics about the annotated data including the final label distributions and interrater reliability are presented. Model hyperparameters are presented and discussed with respect to their mean F1 scores and standard deviations.

Dataset	Total Items	Duplicate Items	Final Item Count	Reviewers
Code Smell Commits	471	30	441	5
Project A Commits	1536	131	1405	6
Project B Commits	238	11	227	5
Project B Comments	469	0	469	5
Project B Docstrings	181	0	181	5

TABLE 1: *Overview of the five datasets, including: (1) counts of original entries, (2) duplicate entries, excluding first occurrence, (3) final entry count with duplicates removed, and (4) number of reviewers that annotated each dataset.*

Dataset	Answer	Annotation	Sub Annotation	Bias
Code Smell Commits	0.85 ± 0.23	0.83 ± 0.28	0.44 ± 0.19	0.22 ± 0.35
Project A Commits	0.86 ± 0.21	0.87 ± 0.24	0.50 ± 0.20	0.39 ± 0.40
Project B Commits	0.78 ± 0.24	0.89 ± 0.24	0.43 ± 0.21	0.35 ± 0.38
Project B Comments	0.91 ± 0.19	0.92 ± 0.20	0.51 ± 0.17	0.43 ± 0.48
Project B Docstrings	0.95 ± 0.15	0.94 ± 0.16	0.51 ± 0.15	0.42 ± 0.49

TABLE 2: *Interrater reliability across the annotated datasets as percentages, with a +/- standard deviation. "Answer" refers to the annotation response type (i.e., accept, reject, ignore). "Annotation" considers the specific bias label, where reject/ignore are empty strings. "Sub-Annotation" considers the subset of entries in which at least one reviewer selected a bias label. "Bias" compares only the bias labels selected by reviewers (i.e., reject/ignore responses are not considered). Reviewers typically agree on whether an entry is biased, but not on the bias type.*

Annotated Datasets

An overview of the four datasets in terms of total number of items, number of duplicate entries, final number of items after accounting for duplicates, and number of reviewers to annotate is provided in Table 1.

To quantify variance in interpretation of bias presentation in software commit messages and comments, interrater reliability was computed based on percent agreement across reviewers. Percent agreement is computed as the number of matching pairs over the number of total possible pairs.

For answer reliability, the number of matching answer pairs (i.e., “accept”, “reject”, or “ignore”) is divided by the total number of possible pairs. For label reliability, we start with the high-level measure of all label options, including the empty label string that results from selection of “reject” or “ignore”. We refer to this measure as annotation reliability, as it accounts for a combination of answer and label selection, though at the cost of instances of “reject” and “ignore” being indistinguishable. Given the expected imbalance of bias versus non-biased entries, we also provide an average of the reliability scores for the subset in which at least one bias label is selected. We refer to this measure as sub-annotation reliability. Lastly, we compute a bias reliability measure in which we compare only the label options available when a reviewer “accepts” an entry as biased.

There were six reviewers for the Project A Commits dataset and five reviewers for all other datasets. Interrater reliability was computed across reviewer annotations and are summarized in Table 2. The distributions of bias labels for each dataset are represented in Figure 3. Overall, reliability measures ranged from 0.78 to 0.91 for answers, 0.83 to 0.92 for annotations, 0.43 to 0.51 for sub-annotations, and 0.22 to 0.43 for bias labels across the four datasets. An overview of the final annotation labels is provided in Table 3.

Given the nature of the data being annotated, we expected a significant amount of variance in how reviewers interpret commit messages and in-source comments, especially without additional context about the relevant code. This was confirmed with the interrater reliability for top-level answers averaging to 85% agreement, while reliability on bias type averaged to 35%. However, we didn’t expect the level of disagreement to be so high, especially when reviewing the label distributions by reviewer. For example, some reviewers used "Other" or selected multiple labels at a much greater rate than others. This may have resulted from the reviewers being unclear on what specific bias was present.

Further, the overall distribution of biased versus not biased entries by dataset supports that artifact types (e.g., comment, commit) are used differently. For example, both comments and docstrings tend to be more technical in nature, with comments typically reflecting procedural knowledge and docstrings describing the purpose, inputs, and outputs of a class or function. This is reflected by all three commit message datasets having approximately 12% of messages flagged as biased, while comments and docstrings only had 6% and 4% biased entries, respectively.

Dataset	Total Items	Rejected (Not Biased)	Accepted (Biased)	Ignored (Excluded)
Code Smell Commits	441	389	51	1
Project A Commits	1,405	1,154	162	89
Project B Commits	227	140	26	61
Project B Comments	469	430	27	12
Project B Docstrings	181	174	7	0

TABLE 3: *Overview of the finalized annotations for each dataset. Entries labeled as "ignore" are excluded from the datasets for subsequent modeling.*

Modeling

Table 4 summarizes the results for each model, along with the best-performing hyperparameters as determined by a parameter sweep. The mean and standard deviation of F1 Scores are computed across three randomized train/test splits within the same dataset.

No models were trained using the dataset comprised of docstrings due to the extreme imbalance in labels (i.e., <5% labeled as bias). The docstring dataset had a total of 7 (of 181) entries labeled as biased. This may be attributed to the inherit technical nature of docstrings, combined with the low quantity of docstrings collected during data curation.

The multi-label model (F1 = 72.1%) did not meet expectations because it consistently predicted that no bias was typically present. This model was over-fit given that the biased entries were now split among four separate bias labels, increasing the level of imbalance. Though this finding may be due to insufficient training data availability, it’s interesting to note that interrater reliability

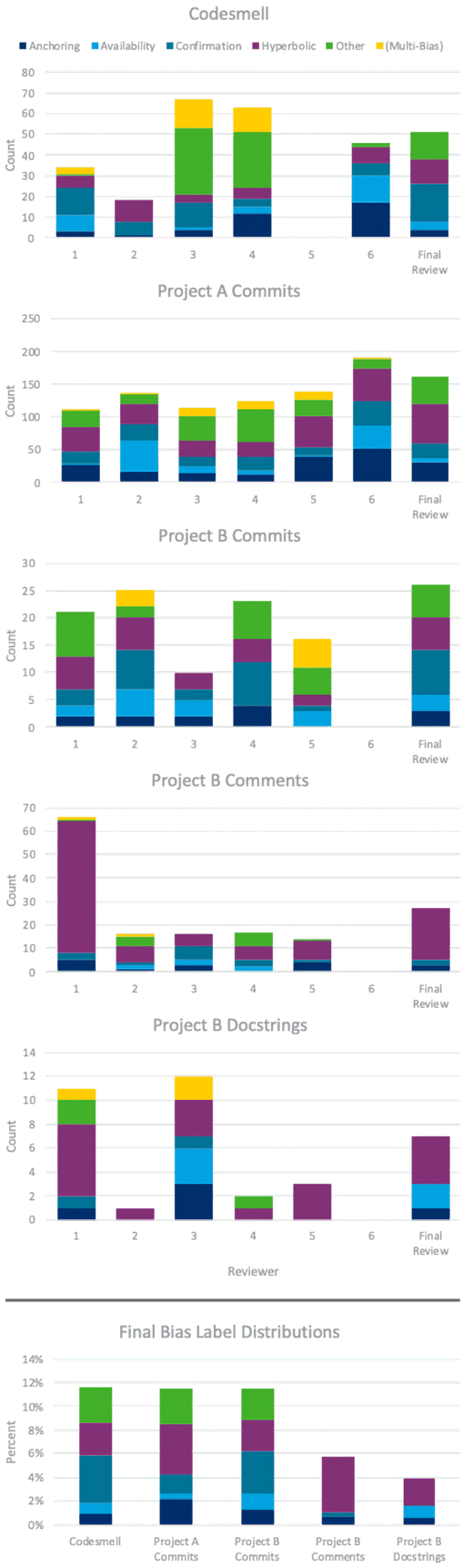

Fig. 3: *Distribution of bias labels per dataset. The first five plots show the distribution of label counts by each reviewer and the finalized review process. The last plot shows the finalized distribution of labels as percentages to standardize visualization across all datasets. The final plot reflects the data used for modeling.*

Dataset	Model Type	Max Samples	Drop Rate	Batch Range	Mean F1	Std. Dev.
Project A Commits	Binary	220	40%	4-64	81.2%	2.6%
Project B Commits	Binary	28	20%	8-64	65.9%	14.0%
Project A + B Commits	Binary	247	20%	4-64	79.0%	5.1%
Project A + B Commits	Multi Label	188	20%	8-32	72.1%	5.8%
Project B Commits + Comments	Binary	104	40%	8-32	78.6%	6.8%
All Internal Data	Binary	324	40%	8-64	82.3%	3.9%

TABLE 4: *Hyperparameters selected and corresponding results for each model. Model Type refers to whether the model predicted Bias vs. No Bias (Binary) or the particular bias types (Multi-Label). "Max Samples" refers to the maximum number of samples allowed for each bias category, to prevent over-fitting given data imbalance. "Drop Rate" and "Batch Range" are hyperparameters for the NLP model. "Mean F1" and "Std. Dev" refer to the model results across three randomized train/test splits within the same dataset.*

follows a similar pattern when defining a specific bias label. The confusion matrices from each of the three instances of this model is Figure 4.

All binary classification models performed in parity with one another, with mean F1 scores ranging from 78.6% to 82.3%. These models performed better than the multi-label models given less data imbalance between the binary categories (i.e., bias vs. no bias). The model trained on Project B Commits data was the only exception, which performed at 65.9%, most likely due to the significantly smaller size of the training dataset.

The best performing model (F1 = 82.3%) was trained using the largest dataset (i.e., the combined commit messages and comments for both Projects A and B) as a binary classification model. The confusion matrices of the three instances of this model is in Figure 5.

Conclusions and Implications

Through this project, two well-curated datasets were generated: one derived from the commit messages of Projects A and B and the other created by labeling an existing collection of code refactoring-related commit messages from various free and open-source software projects[6]. This data is valuable not only because it is the first of its kind, but also because it is representative of technical artifacts generated during the software development process.

The level of variability in bias annotations across reviewers emphasizes the difficulty in discerning whether a statement is biased without insight of the surrounding context. This is further exacerbated when it comes to identifying the type of bias. Furthermore, limiting reviewers to a single annotation per entry may alleviate the risk of reviewers selecting multiple labels when uncertain. Our interrater reliability inherently resulted in lower scores for multi-label annotations. For example, ['ANCHORING,HYPERBOLIC'] and ['HYPERBOLIC'] results in bias reliability of zero even though both reviewers thought hyperbolic discounting was present. The level of variation may also arise from individual differences in writing commit messages and comments; messages that are longer or enumerate each change made are more likely to elicit language suggestive of bias compared to

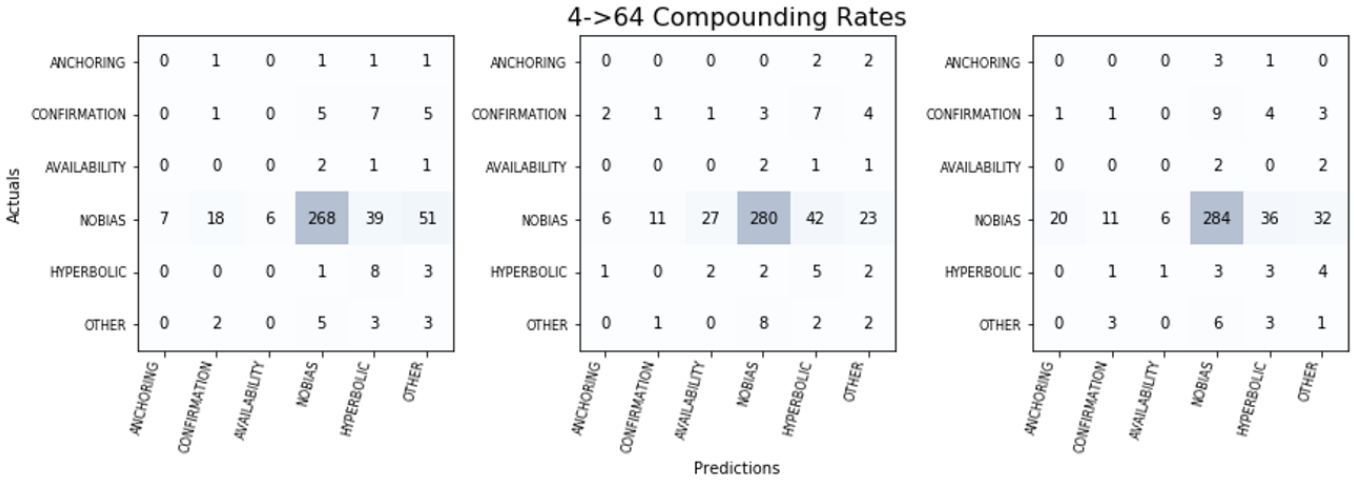

Fig. 4: *Confusion matrices for the multi-label model on "Project A + B Commits". Each matrix is an instance of a model run on a randomized split of the data. All three exemplify over-fitting to the "no bias", attributable to data imbalance. This pattern also mirrors the low interrater reliability scores for specific bias labels.*

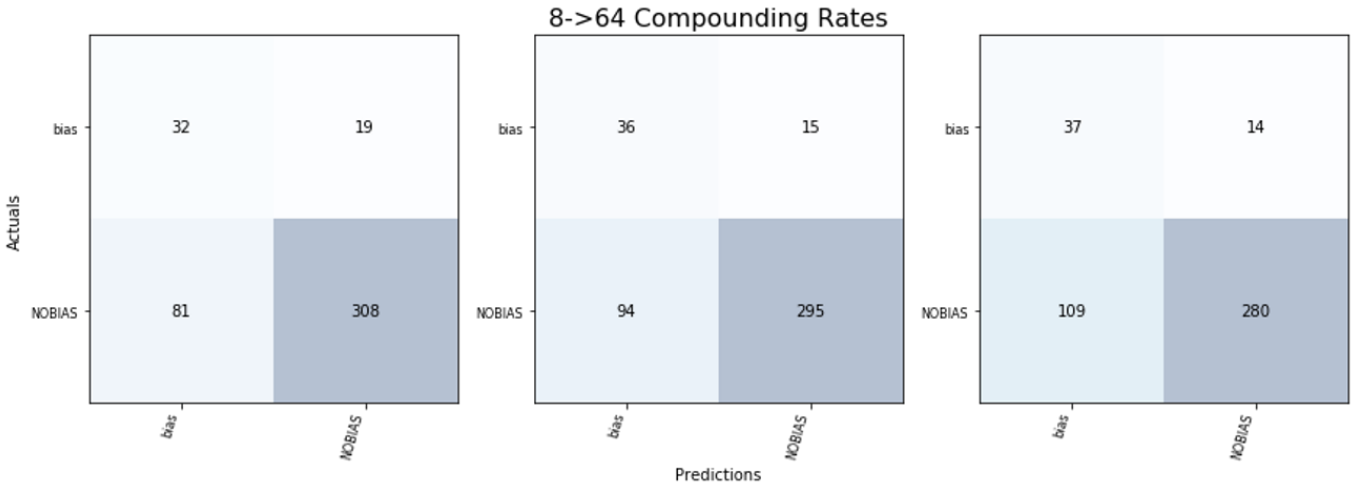

Fig. 5: *Confusion matrices for the binary model on "All Internal Data". Each matrix is an instance of a model run on a randomized split of the data. This model performed the best overall, attributable to the larger dataset size and reduced data imbalance compared to the other models.*

highly concise messages. Properly identifying bias in software artifacts may require consideration for informing software teams on message structuring for consistency and utility.

Possible follow-on efforts to this study will investigate further ways to improve multi-label modeling of bias. The multi-label model was over-trained due to the significant quantity of non-biased data versus the other four categories of bias. This differs from the binary models, which had the advantage of being able to combine those four bias categories together, resulting in a more balanced dataset. Obtaining more data, specifically of entries which are biased, will likely improve model robustness for both the binary and multi-label models.

While data quantity remains an issue, we also note some disagreement in data labels, reflected in the interrater reliability (Table 2). It was not surprising to see the multi-label model struggle to select the correct bias label, as the annotators tended to disagree on which biases were present in specific data points. We had a process to select a single bias label for each entry from the pool of bias labels that the annotators independently selected. It is possible that our model actually agreed with one of the bias labels that an annotator voted on, but was rejected or changed during the final review label review. A follow-on effort to this study will better measure the multi-label model's performance against the pool of bias labels candidates, rather than the single entry selected during annotation review.

Future research efforts that can build on these results include the generation of datasets and models that consider the impact of individual words or short phrases on bias classification, application of a bias detection tool in tracing the source of a significant failure to the engineering process (as opposed to a particular line of code), and investigation of the impact of cognitive bias on code quality metrics. Additionally, larger datasets, especially ones containing in-code comments and document strings, are necessary to quantify the impact of cognitive biases on the quality of finished software systems. In the future, larger projects may require the development of post-mortem reports to identify which aspects of the research, design, and development cycles are most impactful to overall project success or failure. With such data available researchers can begin to answer the central question regarding the impact of individual biases from a holistic perspective.

Acknowledgements

We thank Michael Krein, Lisa Baraniecki, and Owen Gift for their contributions to annotating the datasets used in this effort.

1. M. Delgado-Rodriguez and J. Llorca, *Bias*, Journal of Epidemiology & Community Health, 58(8):635-641, 2004.

2. R. Mohanani, I. Salman, B. Turhan, P. Rodríguez and P. Ralph, *Cognitive biases in software engineering: a systematic mapping study*, IEEE Transactions on Software Engineering, 2018.

3. W. Stacy and J. MacMillan, *Cognitive bias in software engineering*, Communications of the ACM, 38(6):57-63, 1995.

4. G. Calikli and A. Bener, *Empirical analysis of factors affecting confirmation bias levels of software engineers*, Software Quality Journal, 23(4):695-722, 2015.

5. K. Mohan and R. Jain, *Using traceability to mitigate cognitive biases in software development*, Communications of the ACM, 51(9):110-114, 2008.

6. E. AlOmar, M. W. Mkaouer and A. Ouni, *Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages*, IEEE, no. 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), 2019.

How PDFrw and fillable forms improves throughput at a Covid-19 Vaccine Clinic

Haw-minn Lu‡*, José Unpingco‡

Abstract—PDFrw was used to prepopulate Covid-19 vaccination forms to improve the efficiency and integrity of the vaccination process in terms of federal and state privacy requirements. We will describe the vaccination process from the initial appointment, through the vaccination delivery, to the creation of subsequent required documentation. Although Python modules for PDF generation are common, they struggle with managing fillable forms where a fillable field may appear multiple times within the same form. Additionally, field types such as checkboxes, radio buttons, lists and combo boxes are not straightforward to programmatically fill. Another challenge is combining multiple *filled* forms while maintaining the integrity of the values of the fillable fields. Additionally, HIPAA compliance issues are discussed.

Index Terms—acrobat documents, form filling, HIPAA compliance, COVID-19

Introduction

The coronavirus pandemic has been one of the most disruptive nationwide events in living memory. The frail, vulnerable, and elderly have been disproportionately affected by serious hospitalizations and deaths. Notwithstanding the amazing pace of vaccine development, logistical problems can still inhibit large-scale vaccine distribution, especially among the elderly. Vaccination centers typically require online appointments to facilitate vaccine distribution by State and Federal governments, but many elderly do not have Internet access or know how to make online appointments, or how to use online resources to coordinate transportation to and from the vaccination site, as needed.

As a personal anecdote, when vaccinations were opened to all aged 65 and older, one of the authors tried to get his parents vaccinated and discovered that the experience documented here [Let21] was unfortunately typical and required regularly pinging the appointment website for a week to get an appointment. However, beyond persistence, getting an appointment required monitoring the website to track when batches of new appointments were released --- all tasks that require an uncommon knowledge of Internet infrastructure beyond most patients, not just the elderly.

To help San Diego County with the vaccine rollout, the Gary and Mary West PACE (WestPACE) center established a pop-up point of distribution (POD) for the COVID-19 vaccine [pre21] specifically for the elderly with emphasis on those who are most vulnerable. The success in the POD was reported in the local news media [Lit21] [Col21] and prompted the State of California to ask WestPACE's sister organization (the Gary and Mary West Health Institute) to develop a playbook for the deploying a pop-up POD [pod21].

This paper describes the logistical challenges regarding the vaccination rollout for WestPACE and focuses on the use of Python's `PDFrw` module to address real-world sensitive data issues with PDF documents.

This paper gives a little more background of the effort. Next the overall infrastructure and information flow is described. Finally, a very detailed discussion on the use of python and the `PDFrw` library to address a major bottleneck and volunteer pain point.

Background

WestPACE operates a Program of All-Inclusive Care for the Elderly (PACE) center which provides nursing-home-level care and wrap-around services such as transportation to the most vulnerable elderly. To provide vaccinations to WestPACE patients as quickly as possible, WestPACE tried to acquire suitable freezers (some vaccines require special cold storage) instead of waiting for San Diego County to provide them; but, due to high-demand, acquiring a suitably-sized freezer was very problematic. As a pivot, WestPACE opted to acquire a freezer that was available but with excess capacity beyond what was needed for just WestPACE, and then collaborated with the County to use this excess capacity to establish a walk-up vaccination center for all San Diego senior citizens, in or out of WestPACE.

WestPACE coordinated with the local 2-1-1 organization responsible for coordination of community health and disaster services. The 2-1-1 organization provided a call center with in-person support for vaccine appointments and transportation coordination to and from WestPACE. This immediately eased the difficulty of making online appointments and the burden of transportation coordination. With these relationships in place, the vaccination clinic went from concept to active vaccine distribution site in about two weeks resulting in the successful vaccination of thousands of elderly.

Although this is a technical paper, this background describes the real impact technology can make in the lives of the vulnerable and elderly in society in a crisis situation.

Infrastructure

The goal of the WestPACE vaccine clinic was to provide a friendly environment to vaccinate senior citizens. Because this was a non-profit and volunteer effort, the clinic did not have any pre-existing

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

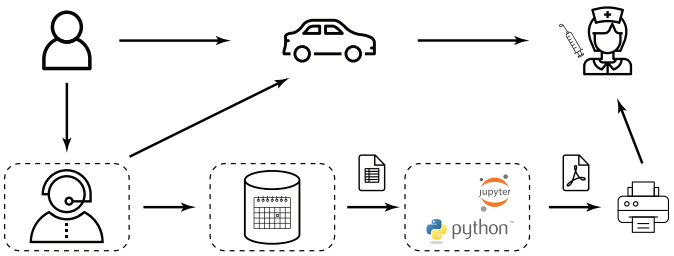

Fig. 1: *Vaccination Pipeline*

record management practices with corresponding IT infrastructure to handle sensitive health information according to Health Insurance Portability and Accountability Act (HIPAA) standards. One key obstacle is paperwork for appointments, questionnaires, consent forms, and reminder cards (among others) that must be processed securely and at speed, given the fierce demand for vaccines. Putting the burden of dealing with this paperwork on the patients would be confusing for the patient and time-consuming and limit the overall count of vaccinations delivered. Thus, the strategy was to use electronic systems to handle Protected Health Information (PHI) wherever possible and comply with HIPAA requirements [MF19] for data encryption at rest and in-transit, including appropriate Business Associate Agreements (BAA) for any cloud service providers [FKR+16]. For physical paper, HIPAA requirements mean that PHI must always be kept in a locked room or a container with restricted access.

Figure 1 shows a high level view of the user experience and information flow. Making appointments can be challenging, especially those with limited caregiver support. Because the appointment systems were set up in a hurry, many user interfaces were confusing and poorly designed. In the depicted pipeline, the person (or caregiver) telephones the 2-1-1 call center and the live operator collects demographic and health information, and coordinates any necessary travel arrangements, as needed. The demographic and health information is entered into the appointment system managed by the California Department of Public Health. The information is then downloaded to the clinic from the appointment system the day before the scheduled vaccination. Next, a forms packet is generated for every scheduled patient and consolidated into a PDF file that is then printed and handed to the volunteers at the clinic. The packet consolidates documents including consent forms, health forms, and CDC-provided vaccination cards.

When the patient arrives at the clinic, their forms are pulled and a volunteer reviews the questions while correcting any errors. Once the information is validated, the patient is directed to sign the appropriate forms. The crucially efficient part is that the patient and volunteer only have to *validate* previously collected information instead of filling out multiple forms with redundant information. This was crucial during peak demand so that most patients experienced less than a five minute delay between arrival and vaccine administration. While there was consideration of commercial services to do the electronic form filling and electronic signatures, they were discounted because these turned out to be too expensive and time-consuming to set up.

Different entities such as 2-1-1 and the State of California handle certain elements of the data pipeline, but strict HIPAA requirements are followed at each step. All clinic communications with the State appointment system were managed through a properly authenticated and encrypted system. The vaccine clinic utilized pre-existing, cloud-based HIPAA-compliant system, with corresponding BAAs. All sensitive data processing occurred on this system. The system, which is described at [HmLAKJU20], uses both python alone and in Jupyter notebooks.

Finally, the processed PDF forms were transferred using encryption to a server at the clinic site where an authorized operator printed them out. The paper forms were placed in the custody of a clinic volunteer until they were delivered to a back office for storage in a locked cabinet, pursuant to health department regulations.

Though all aspects of the pipeline faced challenges, the pre-population of forms turned out to be surprisingly difficult due to the lack of programmatic PDF tools that properly work with fillable forms. The remainder of the paper discusses the challenges and provides instructions on how to use Python to fill PDF forms for printing.

Programmatically Fill Forms

Programmatically filling in PDF forms can be a quick and accurate way to disseminate forms. Bits and pieces can be found throughout the Internet and places like Stack Overflow but no single source provides a complete answer. The *Medium* blog post by Vivsvaan Sharma [Sha20] is a good starting place. Another useful resource is the PDF 1.7 specification [pdf08]. Since the deployment of the vaccine clinic, the details of the form filling can be found at WestHealth's blog [Lu21]. The code is available on GitHub as described below.

The following imports are used in the examples given below.

```
import pdfrw
from pdfrw.objects.pdfstring import PdfString
from pdfrw.objects.pdfstring import BasePdfName
from pdfrw import PdfDict, PdfObject
```

Finding Your Way Around PDFrw and Fillable Forms

Several examples of basic form filling code can be found on the Internet, including the above-mentioned *Medium* blog post. The following is a typical snippet which was taken largely from the blog post.

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

The type of `annotation['/T']` is `pdfString`. While some sources use `[1:-1]` to extract the string from `pdfString`, the `to_unicode` method is the proper way to extract the string. According to the PDF 1.7 specification § 12.5.6.19, all fillable forms use widget annotation. The check for `annotation['/SubType']` filters the annotations to only widget annotations.

To set the value `value`, a `PDFString` needs to be created by encoding `value` with the `encode` method. The encoded `PDFString` is then used to update the `annotation` as shown in the following code snippet.

```
annotation.update(PdfDict(V=PdfString.encode(value)))
```

This converts `value` into a `PdfString` and updates the `annotation`, creating a value for `annotation['/V']`.

In addition, at the top level of the `PdfReader` object `pdf`, the `NeedAppearances` property in the interactive form dictionary, `AcroForm` (See § 12.7.2) needs to be set, without this, the fields are updated but will not necessarily display. To remedy this, the following code snippet can be used.

```
pdf.Root.AcroForm.update(PdfDict(
    NeedAppearances=PdfObject('true')))
```

Multiple Fields with Same Name

Combining the code snippets provides a simple method for filling in text fields, except if there are multiple instances of the same field. To refer back to the clinic example, each patient's form packet comprised multiple forms each with the `Name` field. Some forms even had the `Name` appear twice such as in a demographic section and then in a `Print Name` field next to a signature line. If the code above on such a form were run, the `Name` field will not show up.

Whenever the multiple fields occur with the same name, the situation is more complicated. One way to deal with this is to simply rename the fields to be different such as `Name-1` and `Name-2`, which is fine if the sole use of the form is for automated form filling. This would require access to a form authoring tool. If the form is also to be used for manual filling, this would require the user to enter the `Name` multiple times.

When fields appear multiple times, the widget annotation does not have the `/T` field but has a `/Parent` field. As it turns out this `/Parent` contains the field name `/T` as well as the default value `/V`. Each `/Parent` has one `/Kids` for each occurrence of the field. To modify the code to handle repeated occurrences of a field, the following lines can be inserted:

```
if not annotation['/T']:
    annotation=annotation['/Parent']
```

These lines allow the inspection and modifications of annotations that appear more than once. With this modification, the result of the inspection code yields:

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if not annotation['/T']:
                annotation=annotation['/Parent']
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

With this code in the above example, `Name` would be printed multiple times, once for each instance, but each instance points to the same `/Parent`. With this modification, the form filler actually fills the `/Parent` value multiple times, but this has no impact since it is overwriting the default value with the same value.

Checkboxes

In accordance to §12.7.4.2.3, the checkbox state can be set as follows:

```
def checkbox(annotation, value):
    if value:
        val_str = BasePdfName('/Yes')
    else:
        val_str = BasePdfName('/Off')
    annotation.update(PdfDict(V=val_str))
```

This could work if the export value of the checkbox is `Yes`, which is the default, but not when the export value is something else. The easiest solution is to edit the form to ensure that the export value of the checkbox is `Yes` and the default state of the box is unchecked. The recommendation in the specification is that it be set to `Yes`. In the event tools to make this change are not available, the `/V` and `/AS` fields should be set to the export value not `Yes`. The export value can be inspected by examining the appearance dictionary `/AP` and specifically at the `/N` field. Each annotation has up to three appearances in its appearance dictionary: `/N`, `/R` and `/D`, standing for *normal*, *rollover*, and *down* (§12.5.5). The latter two have to do with appearance in interacting with the mouse. The normal appearance has to do with how the form is printed.

There may be circumstances where the form has checkboxes whose default state is checked. In that case, in order to uncheck a box, the best practice is to delete the `/V` as well as the `/AS` field from the dictionary.

According to the PDF specification for checkboxes, the appearance stream `/AS` should be set to the same value as `/V`. Failure to do so may mean that the checkboxes do not appear.

More Complex Forms

For the purpose of the vaccine clinic application, the filling of text fields and checkboxes were all that were needed. However, for completeness, other form field types were studied and solutions are given below.

Radio Buttons

Radio buttons are by far the most complex of the form entry types. Each widget links to `/Kids` which represent the other buttons in the radio group. Each widget in a radio group will link to the same 'kids'. Much like the 'parents' for the repeated forms fields with the same name, each kid need only be updated once, but the same update can be used multiple times if it simplifies the code.

In a nutshell, the value `/V` of each widget in a radio group needs to be set to the export value of the button selected. In each kid, the appearance stream `/AS` should be set to `/Off` except for the kid corresponding to the export value. In order to identify the kid with its corresponding export value, the `/N` field of the appearance dictionary `/AP` needs to be examined just as was done with the checkboxes.

The resulting code could look like the following:

```
def radio_button(annotation, value):
    for each in annotation['/Kids']:
        # determine the export value of each kid
        keys = each['/AP']['/N'].keys()
        keys.remove('/Off')
        export = keys[0]

        if f'/{value}' == export:
            val_str = BasePdfName(f'/{value}')
        else:
            val_str = BasePdfName(f'/Off')
        each.update(PdfDict(AS=val_str))

    annotation.update(PdfDict(
        V=BasePdfName(f'/{value}')))
```

Combo Boxes and Lists

Both combo boxes and lists are forms of the form type *choice*. The combo boxes resemble drop-down menus and lists are similar to list pickers in HTML. Functionally, they are very similar in form filling. The value `/V` and appearance stream `/AS` need to be set to their exported values. The `/Op` field yields a list of lists associating the exported value with the value that appears in the widget.

To set the combo box, the value needs to be set to the export value.

```
def combobox(annotation, value):
    export=None
    for each in annotation['/Opt']:
        if each[1].to_unicode()==value:
            export = each[0].to_unicode()
    if export is None:
        err = f"Export Value: ""{value} Not Found"
        raise KeyError(err)
    pdfstr = PdfString.encode(export)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

Lists are structurally very similar. The list of exported values can be found in the `/Opt` field. The main difference is that lists based on their configuration can take multiple values. Multiple values can be set with `PDFrw` by setting `/V` and `/AS` to a list of `PdfStrings`. The code presented here uses two separate helpers, but because of the similarity in structure between list boxes and combo boxes, they could be combined into one function.

```
def listbox(annotation, values):
    pdfstrs=[]
    for value in values:
        export=None
        for each in annotation['/Opt']:
            if each[1].to_unicode()==value:
                export = each[0].to_unicode()
        if export is None:
            err = f"Export Value: {value} Not Found"
            raise KeyError(err)
        pdfstrs.append(PdfString.encode(export))
    annotation.update(PdfDict(V=pdfstrs, AS=pdfstrs))
```

Determining Form Field Types Programmatically

While PDF authoring tools or visual inspection can identify each form's type, the type can be determined programmatically as well. It is important to understand that fillable forms fall into four form types, button (push button, checkboxes and radio buttons), text, choice (combo box and list box), and signature. They correspond to following values of the `/FT` form type field of a given annotation, `/Btn`, `/Tx`, `/Ch` and `/Sig`, respectively. Since signature filling is not supported and the push button is a widget which can cause an action but is not fillable, those corresponding types are omitted from consideration.

To distinguish the types of buttons and choices, the form flags `/Ff` field is examined. For radio buttons, the 16th bit is set. For combo box the 18th bit is set. Please note that `annotation['/Ff']` returns a `PdfObject` when returned and must be coerced into an `int` for bit testing.

```
def field_type(annotation):
    ft = annotation['/FT']
    ff = annotation['/Ff']

    if ft == '/Tx':
        return 'text'
    if ft == '/Ch':
        if ff and int(ff) & 1 << 17:  # test 18th bit
            return 'combo'
        else:
            return 'list'
    if ft == '/Btn':
        if ff and int(ff) & 1 << 15:  # test 16th bit
            return 'radio'
        else:
            return 'checkbox'
```

For completeness, the following `text_form` filler helper is included.

```
def text_form(annotation, value):
    pdfstr = PdfString.encode(value)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

This completes the building blocks to an automatic form filler.

Consolidating Multiple Filled Forms

There are two problems with consolidating multiple filled forms. The first problem is that when two PDF files are merged, fields with matching names are associated with each other. For instance, if John Doe were entered in one form's name field and Jane Doe in the second. After combining the two forms John Doe will override the second form's name field and John Doe would appear in both forms. The second problem is that most simple command line or programmatic methods of combining two or more PDF files lose form data. One solution is to "flatten" each PDF file. This is equivalent to printing the file to PDF. In effect, this bakes in the filled form values and does not permit the editing the fields. Going even further, one could render the PDFs as images if the only requirement is that the combined files be printable. However, tools like `ghostscript`, `imagemagick`, and PDFUnite don't do a good job of preserving form data when rendering PDF files.

Form Field Name Collisions

Combining multiple filled PDF files was an issue for the vaccine clinic because the same form was filled out for multiple patients. The alternative of printing hundreds of individual forms was infeasible. To combine a batch of PDF forms, all form field names must be different. Thankfully, the solution is quite simple, in the process of filling out the form using the code above, rename (set) the value of `/T`.

```
def form_filler(in_path, data, out_path, suffix):
    pdf = pdfrw.PdfReader(in_path)
    for page in pdf.pages:
        annotations = page['/Annots']
        if annotations is None:
            continue

        for annotation in annotations:
            if annotation['/SubType'] == '/Widget':
                key = annotation['/T'].to_unicode()
                if key in data:
                    pdfstr = PdfString.encode(data[key])
                    new_key = key + suffix
                    annotation.update(
                        PdfDict(V=pdfstr, T=new_key))
        pdf.Root.AcroForm.update(PdfDict(
             NeedAppearances=PdfObject('true')))
        pdfrw.PdfWriter().write(out_path, pdf)
```

Only a unique suffix needs to be supplied to each form. The suffix can be as simple as a sequential number.

Combining the Files

Solutions for combining PDF files with `PDFrw` can be found on the Internet. The following recipe is typical:

```
writer = PdfWriter()
for fname in files:
    r = PdfReader(fname)
    writer.addpages(r.pages)
writer.write("output.pdf")
```

While the form data still exists in the output file, the rendering information is lost and won't show when displayed or printed. The problem comes from the fact that the written PDF does not have an interactive form dictionary (see §12.7.2 of the PDF 1.7 specification). In particular, the interactive forms dictionary contains the boolean `NeedAppearances` which needs to be set for fields to be shown. If the forms being combined have different interactive form dictionaries, they need to be merged. In this application where the source forms are identical among the various copies, any `AcroForm` dictionary can be used.

After obtaining the dictionary from `pdf.Root.AcroForm` (assuming the `PdfReader` object is stored in `pdf`), it is not clear how to add it to the `PdfWriter` object. The clue comes from a simple recipe for copying a pdf file.

```
pdf = PdfReader(in_file)
PdfWriter().write(out_file, pdf)
```

Examination of the underlying source code shows the second parameter `pdf` to be set to the attribute `trailer` of the `PdfWriter` object. Assuming `acro_form` contains the desired interactive form, the interactive form dictionary can be added to the output document by using `writer.trailer.Root.AcroForm = acro_form`.

Conclusion

A complete functional version of this PDF form filler is open source and can be found at WestHealth's GitHub repository https://github.com/WestHealth/pdf-form-filler. This process was able to produce large quantities of pre-populated forms for senior citizens seeking COVID-19 vaccinations relieving one of the bottlenecks that have plagued many other vaccine clinics.

REFERENCES

[Col21] Annica Colbert. Seniors-only vaccination site. *KPBS News*, Feb 2021. URL: https://www.kpbs.org/podcasts/san-diego-news-now/2021/feb/11/seniors-only-vaccination-site/.

[FKR+16] Barbara L. Filkins, Ju Young Kim, Bruce Roberts, Winston Armstrong, Mark A. Miller, Michael L. Hultner, Anthony P. Castillo, Jean Christophe Ducom, Eric J. Topol, and Steven R. Steinhubl. Privacy and security in the era of digital health: What should translational researchers know and do about it? *American Journal of Translational Research*, 8(3):1560–1580, 2016. Publisher Copyright: © 2016, E-Century Publishing Corporation. All rights reserved.

[HmLAKJU20] Haw-minn Lu, Adrian Kwong, and José Unpingco. Securing Your Collaborative Jupyter Notebooks in the Cloud using Container and Load Balancing Services. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 2 – 10, 2020. doi:10.25080/Majora-342d178e-001.

[Let21] U-T Letters. Opinion: Vaccination frustration grows as seniors weigh limited options. *San Diego Union-Tribune*, Jan 2021. URL: https://www.sandiegouniontribune.com/opinion/story/2021-01-22/vaccination-frustrations-grow-as-seniors-search-for-appointments.

[Lit21] Joe Little. For san diego seniors, making vaccination appointments is as easy as calling 211. *7 San Diego News*, Feb 2021. URL: https://www.nbcsandiego.com/news/local/for-san-diego-seniors-making-vaccination-appointments-is-as-easy-as-calling-211/2531346/.

[Lu21] Haw-minn Lu. Exploring fillable forms with pdfrw, Mar 2021. URL: https://westhealth.github.io/exploring-fillable-forms-with-pdfrw.html.

[MF19] Wilnellys Moore and Sarah Frye. Review of hipaa, part 1: History, protected health information, and privacy and security rules. *Journal of Nuclear Medicine Technology*, 47(4):269–272, 2019. URL: https://tech.snmjournals.org/content/47/4/269, arXiv:https://tech.snmjournals.org/content/47/4/269.full.pdf, doi:10.2967/jnmt.119.227819.

[pdf08] *Document Management - Portable Document Format - Part 1: PDF 1.7*. Adobe Systems Incorporated, 2008.

[pod21] Pop up vaccination point of distribution (pod) for seniors: A how to guide, Apr 2021. URL: https://www.westhealth.org/resource/vaccine-pod-for-seniors/.

[pre21] New covid-19 vaccine site for vulnerable seniors at west pace in san marcos. *West Health Press Releases*, Feb 2021. URL: https://www.westhealth.org/press-release/new-covid-19-vaccine-site-for-vulnerable-seniors-at-west-pace-in-san-marcos/.

[Sha20] Vivsvaan Sharma. Filling editable pdf in python, Aug 2020. URL: https://medium.com/@vivsvaan/filling-editable-pdf-in-python-76712c3ce99.

PyRSB: Portable Performance on Multithreaded Sparse BLAS Operations

Michele Martone[‡*], Simone Bacchio[§]

Abstract—This article introduces **PyRSB**, a Python interface to the **LIBRSB** library. LIBRSB is a portable *performance library* offering so called *Sparse BLAS* (Sparse Basic Linear Algebra Subprograms) operations for modern multicore CPUs. It is based on the *Recursive Sparse Blocks* (**RSB**) format, which is particularly well suited for matrices of large dimensions. PyRSB allows LIBRSB usage with an interface styled after that of **SciPy**'s *sparse matrix* classes, and offers the extra benefit of exploiting multicore parallelism. This article introduces the concepts behind the RSB format and LIBRSB, and illustrates usage of PyRSB. It concludes with a user-oriented overview of speedup advantage of `rsb_matrix` over `scipy.sparse.csr_matrix` running general sparse matrix-matrix multiplication on a modern shared-memory computer.

Introduction

Sparse linear systems solving is one of the most widespread problems in numerical scientific computing. The key to timely solution of sparse linear systems by means of iterative methods resides in fast multiplication of sparse matrices by dense matrices. More precisely, we mean the update: $C \leftarrow C + \alpha AB$ (at the element level, equivalent to $C_{i,k} \leftarrow C_{i,k} + \alpha A_{i,j} B_{j,k}$) where B and C are dense rectangular matrices, A is a *sparse* rectangular matrix, and *alpha* a scalar. If B and C are vectors (i.e. have one column only) we call this operation *SpMV* (short for *Sparse Matrix-Vector product*); otherwise *SpMM* (short for *Sparse Matrix-Matrix product*).

PyRSB [PYRSB] is a package suited for problems where: i) much of the time is spent in SpMV or SpMM, ii) one wants to exploit multicore hardware, and iii) sparse matrices are large (i.e. occupy a significant fraction of a computer's memory).

The PyRSB interface is styled after that of the sparse matrix classes in SciPy [Virtanen20]. Unlike certain similarly scoped projects ([Abbasi18], [PyDataSparse]), PyRSB is restricted to 2-dimensional matrices only.

Background: LIBRSB

LIBRSB [LIBRSB] is a LGPLv3-licensed library written primarily to speed up solution of large sparse linear systems using iterative methods on shared-memory CPUs. It takes its name from the Recursive Sparse Blocks (RSB) data layout it uses. The RSB format is geared to execute multithreaded SpMV and SpMM as fast as possible. LIBRSB is not a solver library, but provides most of the functionality required to build one. It is usable via several languages: C, C++, Fortran, GNU Octave [SPARSERSB], and now Python, too. Bindings for the Julia language have been authored by D.C. Jones [RSB_JL].

LIBRSB has been reportedly used for: Plasma physics [Stegmeir15], sub-atomic physics [Klos18], data classification [Lee15], eigenvalue computations [Wu16], meteorology [Browne15T], and data assimilation [Browne15M].

It is available in pre-compiled form in popular GNU/Linux distributions like Ubuntu [UBUNTU], Debian [DEBIAN], OpenSUSE [OPENSUSE]; this is the best way have a LIBRSB installation to familiarize with PyRSB. However, pre-compiled packages will likely miss compile-time optimizations. For this reason, the best performance will be obtained by bulding on the target computer. This can be achieved using one of the several source-based code distributions offering LIBRSB, like Spack [SPACK], or EasyBuild [EASYBUILD], or GUIX [GUIX]. LIBRSB has minimal dependencies, so even bulding by hand is trivial.

PyRSB [PYRSB] is a thin wrapper around LIBRSB based on Cython [Behnel11]. It aims at bringing native LIBRSB performance and most of its functionality at minimal overhead.

Basic Sparse Matrix Formats

The explicit (**dense**) way to represent any numerical matrix is to list each of its numerical entries, whatever their value. This can be done in Python using e.g. `scipy.matrix`.

```
>>> from scipy import matrix
>>>
>>> A = matrix([[11., 12.], [ 0., 22.]])
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

This matrix has two rows and two columns; it contains three non-zero elements and one zero element in the second row. Many scientific problems give rise to systems of linear equations with many (e.g. millions) of unknowns, but relatively few coefficients which are different than zero (e.g. *<1%*) in their matrix-form representation. It is usually the case that representing these zeroes in memory and processing them in linear algebraic operations does not impact the results, but takes *compute time* nevertheless. In these cases the matrix is usually referred as **sparse**, and appropriate **sparse data structures** and algorithms are sought.

The most straightforward sparse data structure for a numeric matrix is one listing each of the non-zero elements, along with its *coordinate* location, by means of three arrays. This is called **COO**.

* *Corresponding author: michele.martone@lrz.de*
‡ *Leibniz Supercomputing Centre (LRZ), Garching near Munich, Germany*
§ *CaSToRC, The Cyprus Institute, Nicosia, Cyprus*

It's one of the classes in `scipy.sparse`; see the following listing, whose output also illustrates conversion to dense:

```
>>> from scipy.sparse import coo_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> I = [0, 0, 1]
>>> J = [0, 1, 1]
>>> A = coo_matrix((V, (I, J)))
<2x2 sparse matrix of type '<class 'numpy.float64'>'
     with 3 stored elements in COOrdinate format>
>>> B = A.todense()
>>> B
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

Even if yielding the same results, the algorithms beneath differ considerably. To carry out the $C_{i,k} \leftarrow C_{i,k} + \alpha A_{i,j} B_{j,k}$ updates the `scipy.coo_matrix` implementation will get the matrix coefficients from the `V` array, its coordinates from the `I` and `J` arrays, and use those (notice the **indirect access**) to address the operand's elements.

In contrast to that, a dense implementation like `scipy.matrix` does not use any index array: the location of each numerical value (including zeroes) is in direct relation with its row and column indices.

Beyond the `V,I,J` arrays, COO has no extra structure. COO serves well as an exchange format, and allows expressing many operations.

The second most straightforward format is CSR (Compressed Sparse Rows). In CSR, non-zero matrix elements and their column indices are laid consecutively row after row, in the respective arrays `V` and `J`. Differently than in COO, the row index information is compressed in a *row pointers* array `P`, dimensioned one plus rows count. For each row index `i`, `P[i]` is the count of non-zero elements (*nonzeroes*) on preceding rows. The count of nonzeroes at each row `i` is therefore `P[i+1]-P[i]`, with `P[0]==0`. SciPy offers CSR matrices via `scipy.csr_matrix`:

```
>>> import scipy
>>> from scipy.sparse import csr_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> P = [0, 2, 3]
>>> J = [0, 1, 1]
>>> A = csr_matrix((V, J, P))
>>> A.todense()
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

CSR's `P` array allows direct access of each *sparse row*. This helps in expressing row-oriented operations. In the case of the SpMV operation, CSR encourages accumulation of partial results on a per-row basis.

Notice that indices' occupation with COO is strictly proportional to the non-zeroes count of a matrix; in the case of CSR, only the `J` indices array. Consequently, a matrix with more nonzeroes than rows (as usual for most problems) will use less index space if represented by CSR. But in the case of a particularly sparse block of such a matrix, that may not be necessarily true. These considerations back the usage choice of COO and CSR within the RSB layout, described in the following section.

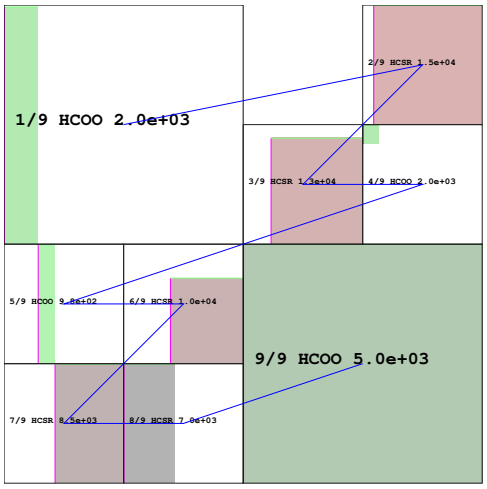

Fig. 1: *Rendering of an RSB instance of classical matrix `bayer02` (sized $14k \times 14k$ with 64k nonzeroes, from the SuiteSparse Matrix Collection [SSMC]); each sparse block is labeled with its own format (the 'H' prefix indicating use of a shorter integer type); each block's effectively non-empty rectangle is shown, in colour; greener blocks have fewer nonzoeroes than average; rosier ones have more. Blocks' rows and columns ranges are highlighted (respectively magenta and green) on the blocks' sides. Note that larger blocks (like `"9/9"`) may have fewer nonzeroes than smaller ones (like `"4/9"`).*

From RSB to PyRSB

Recursive Sparse Blocks in a Nutshell

The Recursive Sparse Blocks (RSB) format in LIBRSB [Martone14] represents sparse matrices by exploiting a hierarchical data structure. The matrix is recursively subdivided in halves until the individual submatrices (also: *sparse blocks* or simply *blocks*) occupy approximately the amount of memory contained in the CPU caches. Each submatrix is then assigned the most appropriate format: COO if very sparse, CSR otherwise.

Any operation on an RSB matrix is effectively a *polyalgorithm*, i.e. each block's contribution will use an algorithm specific to its format, and the intermediate results will be combined. For a more detailed description, please consult [Martone14] and further references from there.

The above details are useful to understand, but not necessary to use PyRSB. To create an `rsb_matrix` object one proceeds just as with e.g. `coo_matrix`:

```
>>> from pyrsb import rsb_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> I = [0, 0, 1]
>>> J = [0, 1, 1]
>>> A = rsb_matrix((V, (I, J)))
>>> A.todense()
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

Direct conversion from `scipy.sparse` classes is also supported. Instancing an RSB structure is computationally more demanding than with COO or CSR (in both memory and time). Exploiting multiple cores and the savings from faster SpMM's shall make the extra construction time negligible.

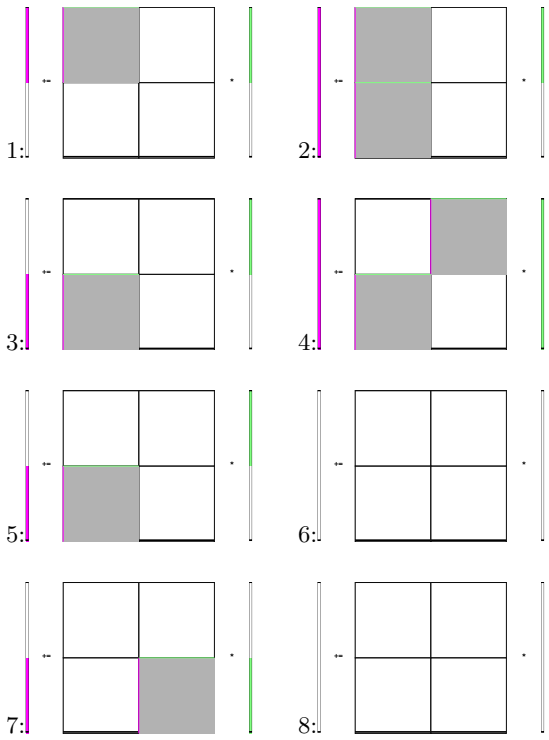

Fig. 2: *SpMV goes through steps leading to the following states: 1) upper left block becomes active; 2) lower left block becomes active; 3) upper left block is done (not active anymore); 4) upper right block becomes active; 5) upper right block is done; 6) lower left block is done; 7) lower right block is now active; 8) lower right block is done.*

Multi-threaded Sparse Matrix-Vector Multiplication with RSB

The following sequence of pictures schematizes eight states of a two-threaded SpMV on an RSB matrix consisting of four (non-empty sparse) blocks. At any moment, up to two blocks are being object of concurrent SpMV (*active*). Here each active block has a gray background; its rows and column ranges are highlighted. Left of the matrix, a (out-of-horizontal-scale) result vector is depicted. For each of the active blocks, the corresponding *active range* (corresponding to the rows) is highlighted on the vector. Similarly, right of the matrix, the (out-of-horizontal-scale) operand vector is shown; its active ranges (corresponding to each blocks' column range) are highlighted.

The idea behind the algorithm is that a thread won't write to a portion of the result array which is currently being updated by another thread. Beyond that, there is no further synchronization of threads.

This algorithm applies to square as well as non-square matrices. It supports transposed operation (in which case the ranges of each block are swapped). Symmetric operation is supported, too; in this case, an additional *transposed* contribution is considered for each block.

As depicted in the first RSB illustration (Fig. 1), the order of the sparse blocks in memory proceeds along a *space-filling curve*. That order of processing the individual blocks can help to deliver data from the memory to the cores faster. For this reason the individual cores attempt to follow that order whenever possible.

To have enough work for each thread, RSB arranges to have more blocks than threads. For this and other trade-offs involved, as well for a formal description of the multiplication algorithm, see [Martone14] and further literature about RSB listed there.

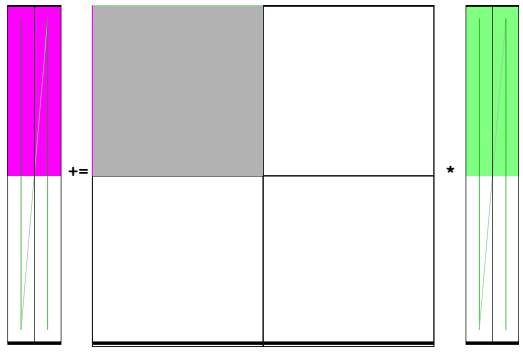

Fig. 3: *A Matrix and its SpMM operands, in* ***columns-major*** *order. Matrix consisting of four sparse blocks, of which one highlighted. Left hand side and right hand side operands consist of two vectors each. These are stored one column after the other (memory follows blue line). Consequently, the two column portions operands pertaining a given sparse block are not contiguous.*

The SpMV algorithm sketched above is what happens *under the hood* in PyRSB. In practice, `rsb_matrix` is used in SpMV just as with `scipy.sparse` classes seen earlier:

```
>>> from numpy import ones
>>> B = ones([2], dtype=A.dtype)
>>> C = A * B
```

Multi-threaded Sparse Matrix-Matrix Multiplication with RSB

With multiple column operands (in jargon, *multiple right hand sides*), the operation result is equivalent to that of performing correspondingly many SpMVs.

In these cases it comes naturally to lay the columns one after the other (consecutively) in memory, and have the resulting *rectangular dense matrix* as operand to the SpMM. Also here the same notation of the previous section is supported; see this example with 2 right hand sides:

```
>>> from numpy import ones
>>> B = ones([2,2], dtype=A.dtype)
>>> C = A * B
```

Let's look at how to deal with this when using the RSB layout. As anticipated, the individual right hand sides may lay after each other, as columns of a rectangular dense matrix. See Fig. 3, where a broken line follows the two operands' layout in memory, also *by columns*.

A straightforward SpMM implementation may run two individual SpMV over the entire matrix, one column at a time. That would have the entire matrix (with all its blocks) being read once per column.

A first RSB-specific optimization would be to run all the per-column SpMVs at a block level. That is, given a block, repeat the SpMVs over all corresponding column portions. This would increase chance of reusing cached matrix elements as the operands are visited. This reuse mechanism is being exploited by LIBRSB-1.2. The *by columns* layout (or *order*) is the recommended one for SpMM there.

The most convenient thing though, would be to read the entire matrix only once. That is the case for LIBRSB-1.3 (scheduled for release in summer 2021): for small column counts, block-level

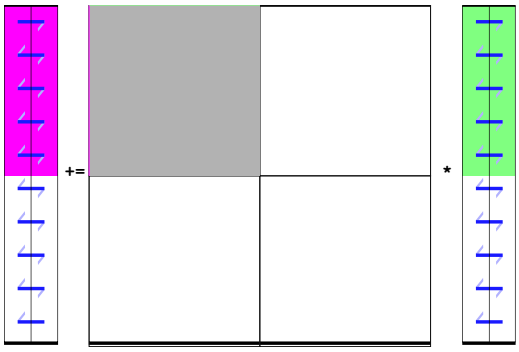

Fig. 4: *A Matrix and its SpMM operands, in* ***rows-major order****. Matrix consisting of four sparse blocks, of which one highlighted. Left hand side and right hand side operands consist of two vectors each, interspersed (memory follows blue line). Consequently, the two column portions operands pertaining a given sparse blocks are contiguous.*

SpMM goes through all the columns while reading a block exactly once.

The aforementioned SpMM algorithm is to be regarded as LIBRSB-specific internals, with not much user-level control over it.

But there is another factor instead, that plays a certain role in the efficiency of SpMM, where the PyRSB user has a choice: the layout of the SpMM operands.

SpMM with different Operands Layout

The **by-columns** layout described earlier and shown in Fig. 3 appears to be the most natural one if one thinks of the columns as laid in successive **multiple arrays**. However, one may instead opt to choose a **by-rows** layout instead, shown in figure 4.

A by-rows layout can be thought as interspersing all the columns, one index at a time. Here in the figure, the blue line follows their **order in memory**. At SpMM time, given one of the input columns, an element at a given index is multiplied by nonzeroes located at that column index. Similarly, given one of the output columns, an element at a given index receives a contribution from the nonzeroes located at that row coordinate. With a by-rows layout of the operands, SpMM may proceed by reading a nonzero once, read all right hand sides at that row index (they are adjacent), and then update the corresponding left hand sides' elements (which are also adjacent). On current cache- and register-based CPUs, the locality induced by this layout leads often to a slightly faster operation than with a by-columns layout.

The by-columns and by-rows layouts go by the respective names of Fortran (`'F'`) and C (`'C'`) order. A user can choose which dense layout to use when creating operands for SpMM. Their physical layouts differ, but NumPy makes their results are interoperable; see e.g.:

```
>>> import scipy, numpy, rsb
>>>
>>> size = 1000
>>> density = 0.01
>>> nrhs = 10
>>>
>>> A = scipy.sparse.random(size, size, density)
>>> A = rsb.rsb_matrix(A)
>>>
>>> B = numpy.random.rand(size, nrhs)
>>>
>>> B_c = numpy.ascontiguousarray(B)
>>> B_f = numpy.asfortranarray(B)
>>>
>>> assert B.flags.c_contiguous
>>> assert B_c.flags.c_contiguous
>>> assert B_f.flags.f_contiguous
>>>
>>> C = A * B
>>> C_c = A * B_c
>>> C_f = A * B_f
```

While both layouts are supported, the `'C'` layout is the recommended one for SpMM operands when using PyRSB with LIBRSB-1.3. Also notice that SpMV is a special case of SpMM with one left-hand side and one right-hand side, so the two layouts are equivalent here. In the following, we will often refer to **right-hand sides count** as by **NRHS**.

Using PyRSB: Environment Setup and Autotuning

Usage of PyRSB requires no knowledge beyond its documentation. However, the underlying LIBRSB library can be configured in a variety of ways, and this affects PyRSB. To begin using PyRSB, a distribution-provided installation shall suffice. To expect best performance results, a *native* LIBRSB build is recommended. The next section comments some basic facts to control LIBRSB and make the most out of PyRSB.

Environment Variables

PyRSB does not use any environment variable directly; it is affected via underlying LIBRSB and Python. By default, LIBRSB it is built with shared-memory parallelism enabled via OpenMP [OPENMP]. As a consequence, a few dozen OpenMP environment variables (all prefixed by `OMP_`) apply to LIBRSB as well. Of these, the most important is the one setting the active threads count: `OMP_NUM_THREADS`. Administrators of HPC (High Performance Computing) systems customarily set this variable to recommended values. Even if unset, chances are good the OpenMP runtime will guess the right value for this. Most other OpenMP variables will be of less use to PyRSB, except one: setting `OMP_DISPLAY_ENV=TRUE` will get current defaults printed at program start (very useful when debugging a configuration).

In addition to the above, there are environment variables affecting specifically LIBRSB. All of those are prefixed by `RSB_`, so to avoid any clash. One recommended to end users is `RSB_USER_SET_MEM_HIERARCHY_INFO`, and is used to override cache hierarchy information detected at runtime or *hard-coded* at build time. Essentially, one can use it to force a finer or coarser blocking. For its usage, and for verification of further LIBRSB defaults, please see its documentation (accessible from [LIBRSB]). Modifying the variables mentioned in this section will be mostly useful on very new or not fully configured systems, or for tuning a bit over the defaults.

RSB Autotuning Procedure for SpMM

Cores count, cache sizes, operands data layout, and matrix structure all play a role in RSB performance. The default blocks layout chosen when assembling an RSB instance may not be the most efficient for the particular SpMM to follow. In practice, given an RSB instance and an SpMM context (vector and scalar operands info, transposition parameter, run-time threads count), it may be the case that a better-performing layout can be found by exploring slightly *coarser* or *finer* blockings, An automated (*autotuning*)

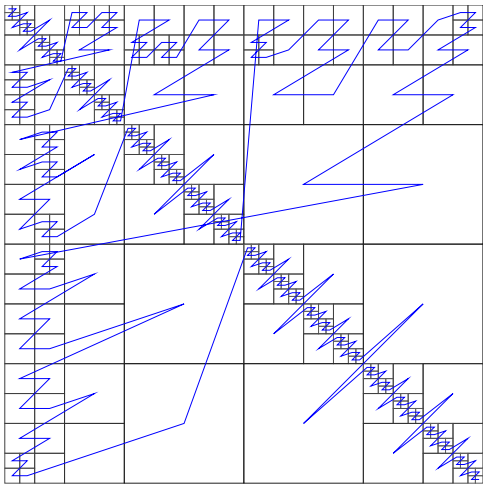

Fig. 5: *Rendering of an RSB instance matrix* `audikw_1` *(for this and other matrices, see table) as* `dtype=numpy.float32` *(or S) after* `autotune(order='C',nrhs=1)` *on our setup. Autotuning merged an initial 766 blocks guess into 295, bringing a* 1.56× *speedup to* `rsb_matrix` *SpMV time. With* `rsb_matrix` *it now takes 1/34th of (1-threaded)* `csr_matrix` *time; before autotuning, it took 1/22th. Autotuning itself took the time of 1.5* `csr_matrix` *SpMV iterations, or 34 pre-autotuning* `rsb_matrix` *SpMV iterations.*

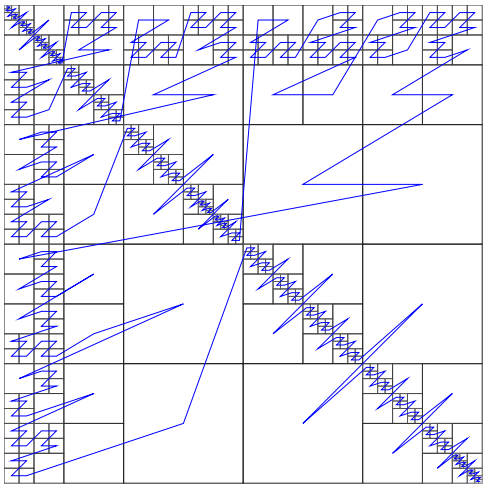

Fig. 6: *Same matrix as Fig. 5, but autotuned with* `nrhs=2`*. Here the initial 766 blocks have been merged into 406, with* 1.14× *speedup. Before autotuning, it took 1/22th of a (1-threaded)* `csr_matrix` *time; now it's 1/31th. Here too, it took the time of 1.5* `csr_matrix` *SpMM iterations, or 34 with the pre-autotuning* `rsb_matrix` *instance.*

procedure for this exists and is accessible via `autotune`. The following example shows how to use it on matrix `audikw_1` from [SSMC].

```
>>> import sys, rsb, numpy
>>> dtype=numpy.float32
>>>
>>> A = rsb.rsb_matrix("audikw_1.mtx",dtype=dtype)
>>> print(A) # original blocking printed out
>>> sf = A.autotune(verbose=False)
>>> print("autotune speedup for SpMV  : %.2e x" %sf )
>>> print(A) # updated blocking printed out
>>>
>>> A = rsb.rsb_matrix("audikw_1.mtx",dtype=dtype)
>>> print(A) # original blocking printed out
>>> sf = A.autotune(verbose=False, transA='N',
>>>       order='C', nrhs=8)
>>> print("autotune speedup for SpMM-8: %.2e x" %sf )
>>> print(A) # updated blocking printed out
```

In scenarios where SpMM is to be iterated many times, time spent autotuning an instance shall amortize over the now faster iterations. See the comments of instances of autotuning on Fig. 5, Fig. 6. and Fig. 7 for realistic use cases.

The reader impatient to see further speedup figures achievable by `autotune` can already peek at Fig. 10.

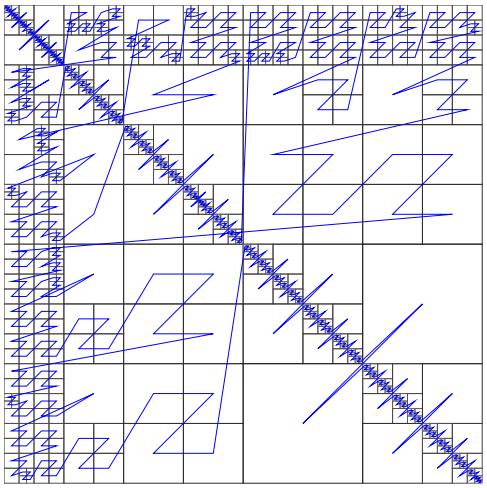

Fig. 7: *Differently than with* `nrhs=1` *or* `nrhs=2`*,* `autotune(nrhs=8)` *did not find a better blocking than the original 766 blocks. Still, the procedure costed the time of 11* `csr_matrix` *SpMM's, or 234* `rsb_matrix` *ones. Though not autotuned, (threaded) RSB takes merely 1/22th the time of CSR here.*

Experiments with SpMM and Autotuning

Purpose of this section is to present **statistics of speedups** one may encounter by using PyRSB instead of SciPy CSR in practical usage. In our choice of experiments, and in the exposition, we favour **breadth** over depth. So **differently than in a paper with HPC in focus**, we focus on the achievable speedup, and not on performance. We also take **shortcuts** which we would not take otherwise, like mixing statistics from *single precision* computations with *double precision* ones, or real-valued and complex-valued ones. Also the very focus of the article, namely comparing directly **threaded RSB to serial CSR** in SciPy would be ill-posed, were we interested to compare the parallelism grade of the two implementations. On the plots that will follow, samples are grouped by matrix; for each one, a *five-number summary* (minimum and maximum, first quartile, second (median) and third quartiles) is drawn with a *boxes and whiskers* representation.

Experimental Setup

We use a *AMD EPYC 7742* node with 64 cores. Scaling of memory bandwidth in STREAM-like loops here is around 10×. Considering we are dealing with memory-bound operations, we chose `OMP_NUM_THREADS=24`, `OMP_PROC_BIND=spread`, and `OMP_PLACES=cores`. `RSB_USER_SET_MEM_HIERARCHY_INFO` was set to `"L2:4/64/16000K,L1:8/64/32K"`. We use CSR from `csr_matrix` in SciPy `e171a1` from Feb 20, 2021, PyRSB `8a6d603` from Jun 08, 2021, pre-release LIBRSB-1.3. For both, we use `-Ofast -march=native -mtune=native` flags and `gcc version 10.2.1 20210110 (Debian 10.2.1-6)`. We use matrices which were also used in

[Martone14], available from https://sparse.tamu.edu/ ([SSMC]); see the table below. Many of these are symmetric; differently than `rsb_matrix`, `csr_matrix` does not support *symmetric SpMM*; therefore in both cases we expand their symmetry and perform only *unsymmetric* (general) SpMM. Before starting any measurement, we run `autotune` on a temporary matrix to *warm-up* the OpenMP environment, once. Then we do one non-timed *warm-up* SpMM before iterating for 0.2s and taking the fastest sample. We repeat this for each of the 28 matrices, right-hand-sides (NRHS) in `1,2,4,8`, order among `'C'` and `'F'`, *BLAS numerical types* in `C,D,S,Z`. When using `rsb_matrix`, we measure both non-autotuned, and autotuned with `autotune(nrhs=...,order=...,tmax=0)`. So the above totals to $28 \cdot 4 \cdot 2 \cdot 4 = 896$ records with samples in SpMM and tuning timing. To avoid also timing repeated allocation of the SpMM result (`C` in `C=A*B`), we allocate it once, and then instead of the $\star$ operator, we use the functions underneath it, which take `C` as argument (**this can be of interest to many performance-conscious users**).

	matrix	nonzeroes	rows	ratio
1	arabic-2005	6.40e+08	2.27e+07	28.1
2	audikw_1	7.77e+07	9.44e+05	82.3
3	bone010	7.17e+07	9.87e+05	72.6
4	channel-500x100x100-b050	8.54e+07	4.80e+06	17.8
5	Cube_Coup_dt6	1.27e+08	2.16e+06	58.8
6	delaunay_n24	1.01e+08	1.68e+07	6.0
7	dielFilterV3real	8.93e+07	1.10e+06	81.0
8	europe_osm	1.08e+08	5.09e+07	2.1
9	Flan_1565	1.17e+08	1.56e+06	75.0
10	Geo_1438	6.32e+07	1.44e+06	43.9
11	GL7d19	3.73e+07	1.91e+06	19.5
12	gsm_106857	2.18e+07	5.89e+05	36.9
13	hollywood-2009	1.14e+08	1.14e+06	99.9
14	Hook_1498	6.09e+07	1.50e+06	40.7
15	HV15R	2.83e+08	2.02e+06	140.3
16	indochina-2004	1.94e+08	7.41e+06	26.2
17	kron_g500-logn21	1.82e+08	2.10e+06	86.8
18	Long_Coup_dt6	8.71e+07	1.47e+06	59.2
19	nlpkkt160	2.30e+08	8.35e+06	27.5
20	nlpkkt200	4.48e+08	1.62e+07	27.6
21	nlpkkt240	7.74e+08	2.80e+07	27.7
22	relat9	3.90e+07	1.24e+07	3.2
23	rgg_n_2_23_s0	1.27e+08	8.39e+06	15.1
24	rgg_n_2_24_s0	2.65e+08	1.68e+07	15.8
25	RM07R	3.75e+07	3.82e+05	98.2
26	road_usa	5.77e+07	2.39e+07	2.4
27	Serena	6.45e+07	1.39e+06	46.4
28	uk-2002	2.98e+08	1.85e+07	16.1

SpMM Speedup: from `csr_matrix` to `rsb_matrix`

Figure 8 summarizes the speed ratio of non-autotuned `rsb_matrix` over `csr_matrix`. Speedup without RSB autotuning ranges from 4× to 64×, with median 15×. Half of observed speedup cases falls between 11× and 20×. A *streaming memory access* benchmark we ran on this machine scaled up to circa 10×, which just less than the observed median speedup (remember `rsb_matrix` is running with multiple cores, but `csr_matrix` cannot exploit that).

For the reader who is not practical of SpMM performance: the memory access pattern of SpMM is typically very irregular, and largely dependent on the sparsity structure of the matrix. For this reason, for most layouts the multicore scaling of SpMM performance (in particular SpMV) tends to be worst than a streaming memory access scaling. But here we are comparing speed ratios of different algorithms, and these ratios differ as well. That reflects

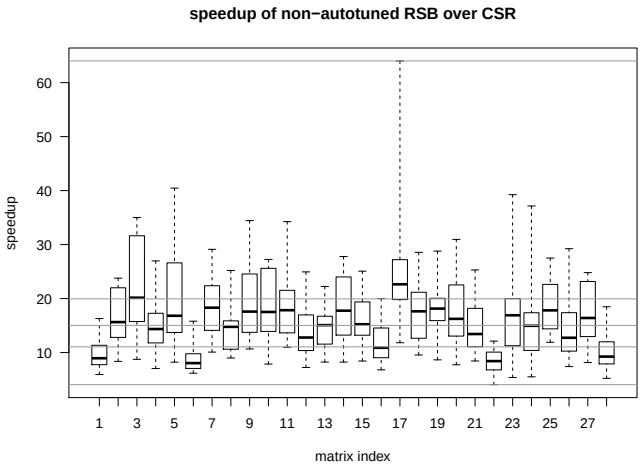

Fig. 8: *Performance samples grouped by matrices. Each box represents a group of measurements on the different numerical type, NRHS, and operands layout. The middle horizontal line is the median speedup of RSB vs CSR, corresponding to* 15×. *The other lines are the extremes, and the first and third quartiles in between (the second quartile being the median value). Notice* autotuned *results in Fig. 9 improve this further.*

the better or worse aptness of a given format to a given matrix. For instance, matrix 17 has nonzeroes scattered quite regularly over the entire matrix, not much clustered: this favours RSB and the *cache blocking* induced by its structure rather than CSR (serial or not). Conversely, matrix 9 has most of its nonzeroes adjacent to some other, which is more CSR-friendly, and a contribution to the lesser improvement when switching to RSB here. See [Martone14] for more RSB-vs-CSR commentary.

The speedups shown so far and those in Fig. 8 rely on default RSB layouts. As said earlier, the RSB format is suited best to scenarios with large matrices and repeated SpMM applications. These are also the scenarios where the usage of `autotune`, which refines the default layout according to the operands at hand, is most convenient.

Figure 9 shows results with autotuned instances. Here `autotune` has been called for each combination of matrix, operands layout, NRHS, and numerical type. The median speedup over CSR here (circa 28.8×) is almost twice the one before autotuning.

With respect to non-autotuned RSB samples, the application of `autotune` brought a median improvement of 1.6×. This includes all samples, inclusive of the lower quartile, with speedup between 1× (no speedup) and 1.2×, which we nevertheless regard as *ineffective* (see next subsection's discussion). An overview of which matrix benefited more, and which less from autotuning is given by Fig. 10. There is no clear trend to see here. We observe that most of the cases (70%) benefited from autotuning. It's worth mentioning that the longer the time limit chosen to run SpMM before taking each performance sample, the less the fluctuation we would have encountered here, and times we chose were quite tight.

Speedups of tuned RSB vs CSR have median 29× with the `'C'` layout, and 28.6× with `'F'` layout; also within RSB the `'C'` layout performs a few percentage points better than `'F'`.

As seen in this section, autotuning can speedup RSB a further bit, but not always. The next section quantifies the cost of autotun-

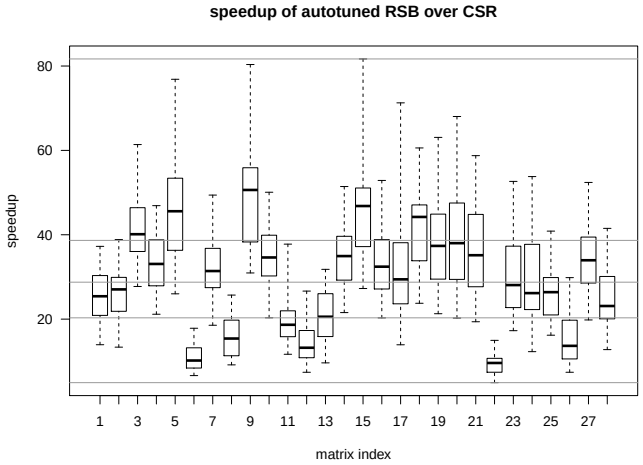

Fig. 9: *We observe speedup over CSR from a few up to* 81.7×*, with median of* 28.8×*. Certain matrices benefit from RSB more (see matrices 5, 9, 15, 18), while others less (6,22,..). Compare the relevant improvement over non-autotuned results in Fig. 8, or see Fig. 10 for the per-matrix ratios.*

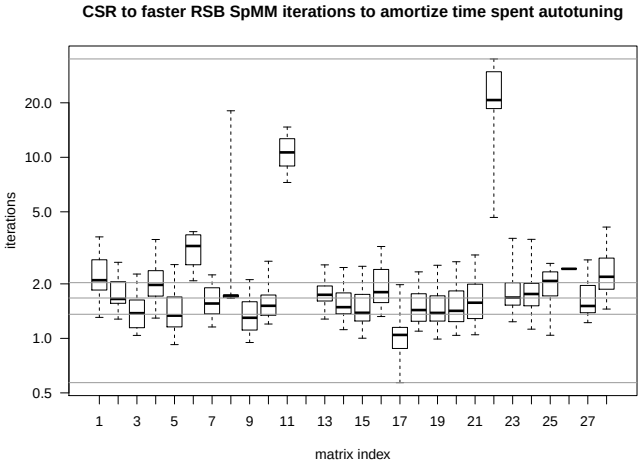

Fig. 11: *Were one to use RSB instead of CSR, and obtain an autotuned instance via* `autotune`*, then this would amortize in few iterations. Notice than in the intended scenarios, where thousands of SpMM are foreseen, this is completely negligible. Note: autotuning was effective in 70% of the cases, represented here and in Fig. 12.*

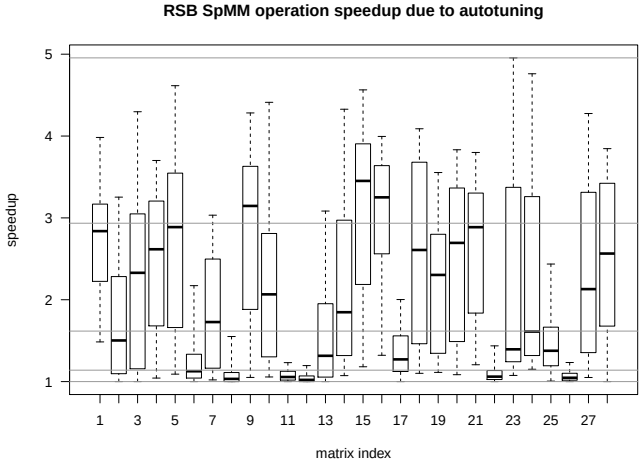

Fig. 10: *Per-sample autotuning effectiveness statistics: autotuned RSB SpMM speed to non-autotuned one. Half of the cases improve by* $> 1.6\times$*, 25% of the cases by* $> 2.9\times$*. Matrices 8,11,12,22,26 seem to barely profit from it. These are the same ones that exhibit the highest ineffective autotuning cost on Fig. 13.*

ing in practical terms, for both effective and ineffective outcomes.

The Cost of RSB Autotuning

As introduced earlier, `autotune` adapts the structure of an RSB matrix, seeking instances which execute a specified operation (here, SpMM) faster. A consistent fraction of the autotuning time is spent measuring SpMM timings of *prospective RSB instances.* It's important to remark: what one wants here is not merely faster execution of SpMM *after* autotuning. What one wants is that autotuning plus all following SpMM iterations shall take less time than the same count of iterations with a non-autotuned matrix. In other words, if the time savings of faster SpMM's cannot cover the autotuning duration, autotuning time is lost. For this reason it is convenient to quantify the number of iterations to reach the first SpMM bringing actual time saving (*amortization*); this is the duration of `autotune` divided by the time *saved* at each iteration (that is, *slow* time with *old RSB blocking*, minus *faster* time with *new RSB blocking*).

For the purpose of this article, we chose to declare autotuning as *effective* if it brings a speedup of 20% or more. With this threshold set, while 94.5% of the cases get *some* speedup, it is 70% that qualify also as effective.

What one observes among effectively autotuned cases (see Fig. 11) is that in 75% of those cases, merely 2.5 CSR iterations are enough to amortize the autotuning time. This is thanks to the large speedup going from (serial) CSR to (parallel) RSB.

If as cost unit we consider going from non-autotuned to autotuned RSB instead, then the relative gain is less (because threaded non-autotuned RSB is already much faster than serial CSR), and consequently, it takes more to amortize it; see Fig. 12.

When autotuning was ineffective (30% of the cases with our $1.2\times$ threshold, though only 5.5% exhibit no speedup at all), we regard its time as lost; in our test setup this was from a few dozen to a few hundred RSB iterations, with median 33; see Fig. 13. If expressed in terms of serial CSR iterations, these would be < 2.8 iterations in half of the cases, < 8 in 75% of the cases.

These results shall convince users that using `autotune` is a good option most of the times.

Conclusions and Future Work

Full utilization of the parallelism potential is important in achieving efficient operations on current CPUs. **PyRSB** does that by giving Python users transparent access to the shared-memory parallel *performance library* **LIBRSB**. Differently than classes in current `scipy.sparse`, but with a very similar usage interface, PyRSB's `rsb_matrix` readily exploits shared-memory parallelism. This article's results section gave a wide sample of speedup statistics with respect to SciPy's `csr_matrix`, on the SpMM operation. Observed median speedup with respect to `csr_matrix` exceeded the known memory bandwidth speedup on the machine; with autotuning, it doubled that, speaking for the good implementation in LIBRSB. Trade-off considerations in

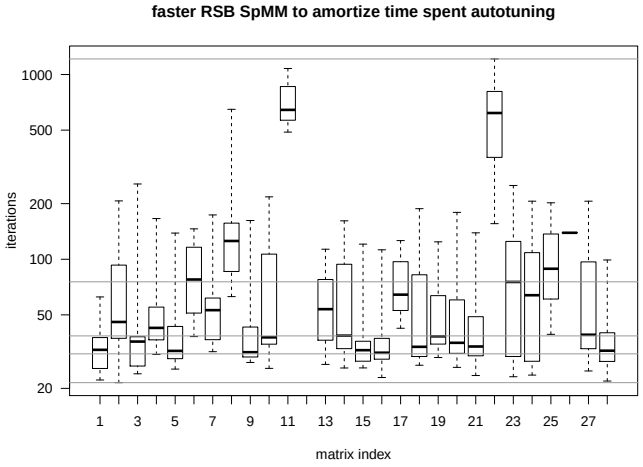

Fig. 12: *If one were to start autotuning from RSB (thus with less improvement potential than with CSR), the amortization times cost more iterations (here, median is* 38.4×*, 75% of the cases below* 76×*). Nevertheless, for many problems, where thousands of iterations are foreseen, this is perfectly acceptable.*

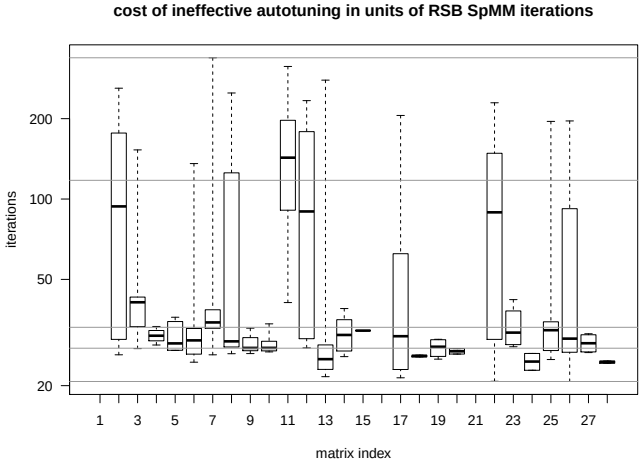

Fig. 13: *There is no guarantee autotuning improves SpMM performance. Actually, autotuning would be unnecessary, if we were able to guess blockings optimal under all circumstances. Indeed, without further analysis, one may even speculate that the default RSB blocking matrices where autotuning was ineffective, was also the* best*. In our experiment, ineffective autotuning searches* ***cost*** 33× *RSB (only* 2.8× *CSR) SpMM iterations in the median case. Note that for certain matrices (1,16,21) autotuning was always effective: this is why these have no associated box here.*

using PyRSB effectively by means of autotuning have also been delineated.

SpMM and autotuning are the *workhorses* of PyRSB and we addressed their use here. Follow-up studies may address or reflect improvements on the LIBRSB side, special use cases, as well as mostly usability-related aspects on the PyRSB side, especially in striving for SciPy interoperability in the user interface. Comparing symmetric SpMM of PyRSB to that of specific *symmetric formats* in SciPy may also be of interest.

Acknowledgments

This work has been financed by **PRACE-6IP**, under Grant agreement ID: 823767, under Project name *LyNcs*. LyNcs is one of 10 collaborations supported by PRACE-6IP, WP8 *"Forward Looking Software Solutions"*. Performance results have been obtained on systems in the test environment **BEAST** (*Bavarian Energy Architecture & Software Testbed*) at the Leibniz Supercomputing Centre. We are grateful to reviewer Meghann Agarwal for her energetic help in improving the quality of this article.

REFERENCES

[PYRSB] *PyRSB*. (2021, May). Retrieved May 28, 2021, https://github.com/michelemartone/pyrsb

[LIBRSB] *LIBRSB*. (2021, May). Retrieved May 28, 2021, https://librsb.sf.net

[Martone14] Michele Martone. "Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the Recursive Sparse Blocks format". Parallel Comput. 40(7): 251-270 (2014). http://dx.doi.org/10.1016/j.parco.2014.03.008

[Virtanen20] P.Virtanen, R.Gommers, T.Oliphant, et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". Nat Methods 17, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2

[Behnel11] S.Behnel, R.Bradshaw, C.Citro, L.Dalcin, D.S.Seljebotn and K.Smith. "Cython: The Best of Both Worlds", in Computing in Science & Engineering, vol. 13, no. 2, pp. 31-39, March-April 2011, doi: https://doi.org/10.1109/MCSE.2010.118

[RSB_JL] *RecursiveSparseBlocks.jl*, (2021, April 08). Retrieved April 08, 2021, from https://github.com/dcjones/RecursiveSparseBlocks.jl.git

[Abbasi18] H.Abbasi, "Sparse: A more modern sparse array library", Proceedings of the 17th Python in Science Conference (SciPy 2018), July 9-15, 2018, Austin, Texas, USA. http://conference.scipy.org/proceedings/scipy2018/hameer_abbasi.html

[PyDataSparse] *PyDataSparse.jl*, (2021, April 08). Retrieved April 08, 2021, from https://github.com/pydata/sparse.

[Lee15] M.Lee, W.Chiang and C.Lin, "Fast Matrix-Vector Multiplications for Large-Scale Logistic Regression on Shared-Memory Systems," 2015 IEEE International Conference on Data Mining, Atlantic City, NJ, USA, 2015, pp. 835-840, doi: https://doi.org/10.1109/ICDM.2015.75

[Stegmeir15] A.Stegmeir (Jan 2015). "GRILLIX: A 3D turbulence code for magnetic fusion devices based on a field line map". Available from INIS: http://inis.iaea.org/search/search.aspx?orig_q=RN:46119630

[Klos18] P.Klos, S.König, H.-W.Hammer, J.E. Lynn, and A.Schwenk. "Signatures of few-body resonances in finite volume". Phys. Rev. C 98, 034004 – Published 24 September 2018, doi: https://doi.org/10.1103/PhysRevC.98.034004

[Wu16] L.Wu. "Algorithms for Large Scale Problems in Eigenvalue and Svd Computations and in Big Data Applications" (2016). Dissertations, Theses, and Masters Projects. Paper 1477068451. http://doi.org/10.21220/S2S880

[Browne15T] P.A. Browne, P.J. van Leeuwen. "Twin experiments with the equivalent weights particle filter and HadCM3". Quarterly Journal of the Royal Meteorological Society, vol. 141, no. 693, pp. 3399-3414, https://doi.org/10.1002/qj.2621

[Browne15M] P.A. Browne, S. Wilson. "A simple method for integrating a complex model into an ensemble data assimilation system using MPI". Environmental Modelling & Software, vol. 68, pp. 122-128, https://doi.org/10.1016/j.envsoft.2015.02.003

[SPACK] *Spack*. (2021, May). Retrieved May 28, 2021, https://spack.io

[EASYBUILD] *EasyBuild*. (2021, May). Retrieved May 28, 2021, https://easybuild.io

[DEBIAN] *Debian*. (2021, May). Retrieved May 28, 2021, http://www.debian.org

[UBUNTU] *Ubuntu*. (2021, May). Retrieved May 28, 2021, http://www.ubuntu.com

[OPENSUSE] *OpenSUSE*. (2021, May). Retrieved May 28, 2021, from https://www.opensuse.org

[GUIX] *GuixHPC*. (2021, May). Retrieved May 28, 2021, from https://hpc.guix.info/

[SPARSERSB] *SparseRSB*, (2021, April 09). Retrieved April 09, 2021, from https://octave.sourceforge.io/sparsersb/

[SSMC] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1 (December 2011), 25 pages. doi: https://doi.org/10.1145/2049662.2049663

[OPENMP] *OpenMP*, (2021, May). Retrieved May 28, 2021, from https://www.openmp.org/

Classification of Diffuse Subcellular Morphologies

Neelima Pulagam[‡], Marcus Hill[‡], Mojtaba Fazli[‡], Rachel Mattson[§], Meekail Zain[‡], Andrew Durden[‡], Frederick D Quinn[¶], S Chakra Chennubhotla[‖], Shannon P Quinn[‡*,**]

Abstract—Characterizing dynamic sub-cellular morphologies in response to perturbation remains a challenging and important problem. Many organelles are anisotropic and difficult to segment, and few methods exist for quantifying the shape, size, and quantity of these organelles. The OrNet (Organelle Networks) framework models the diffuse organelle structures as social networks using graph theoretic and probabilistic approaches. Specifically, this architecture tracks the morphological changes in mitochondria because its structural changes offer insight into the adverse effects of pathogens on the host and aid the diagnosis and treatment of diseases; such as tuberculosis. The OrNet framework offers a segmentation pipeline to preprocess confocal imaging videos that display various mitochondrial morphologies into social network graphs. Earlier methods of anomaly detection in organelle structures include manual identification by researchers in the biology domain. Although those approaches were successful, manual classification is time consuming, tedious, and error-prone. Existing convolutional architectures do not have the capability to adapt to general graphs and fail to represent diffuse organelle morphologies due their amorphous characteristic. Thus, we propose the two different methods to perform classification on these organelles that captures their dynamic behaviors and identifies the fragmentation and fusion of mitochondria. One is a graph deep learning architecture, and the second is an approach that finds a graph representation for each social network and uses a traditional machine learning method for classification. Recent studies have demonstrated graph neural network models perform well on time-series imaging tasks, and the graph architectures are better able to represent amorphous and spatially diffuse structures such as mitochondria. Alternatively, much research has established traditional machine learning methods to be promising and robust models. Testing and comparing different architectures and models will effectively improve the robustness of categorizing distinct structural changes in subcellular organelle structures that is very useful for identifying infection patterns, offering a new way to understand cellular health and dynamic responses.

Introduction

Automation of cell classification remains to be a challenging but very important problem that offers significant benefits to immunology and biomedicine. Specifically, classification of sub cellular perturbations can help characterize healthy cells from infected cells. Morphological changes of sub cellular organelles play a vital role in providing insight into infection patterns [RJCC19].

Tuberculosis (TB), a bacterial disease that mainly infects the lungs, causes structural changes to the mitochondria of cells that have been infected. [DSF13].This ancient disease, although curable and treatable, can be fatal when not diagnosed properly and remains to be the world's leading infectious disease killer having claimed 1.4 million lives in 2019 alone. Each following year brings rising cases of drug resistant TB, with more than 10 million people falling ill with active TB each year. A deeper understanding of the pathogenic processes associated with new infections will allow for the development of effective drug regimens. Although much research has been conducted on the disease, there are many questions on the mechanisms of pathogenesis that remain unanswered. Automating the process of classification offers a faster way to study the rising number of mutations of the Mtb pathogen which will help with the development of treatment plans and vaccines [DSF13].

Recent advancements in fluorescence microscopy and biomedical imaging have offered new ways to analyze these pathogens and their effects on cell health [ADATLBR+18]. Previous studies have proposed artificial intelligence based cell classifiers using convolutional neural networks [OHL+19] , [YRS19]. These networks had a few shortcomings due to the diversity of cells in any system and the studies in modeling different biological phenomena have been disproportionate. Most segmentation tasks deal with morphologies of cells and nuclei. These structures are much easier to segment, model and track than spatially diffuse structures such as mitochondria. Mitochondria act as significant signaling platforms in the cell whose dynamics modulate in response to pathogens to maintain their environment [RJCC19]. Infections induce mitochondrial changes and automating the classification of these anomalies will lead to more knowledge on the morphological changes which can further help create targeted therapies.

We propose two methods to classify mitochondria based on their dynamics by representing the subcellular structures as social network graphs. Graphs offer an effective way to represent the amorphous mitochondrial structures and capture the different spatial morphologies. Furthermore, machine learning on graphs is becoming a very relevant and ubiquitous task that has made significant contributions to deep learning, helping find solutions to several problems in the biomedicine domain.

We analyze the cells of the last frame of the video data that portray the cells after the fusion or fission event to classify which structural change has occurred. We explore two methods that utilize graph machine learning and have proven to be effective in characterizing morphological events given only the last frame

‡ *Department of Computer Science, University of Georgia, Athens, GA 30602 USA*
§ *Department of Cognitive Science, 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*

of the video. The first involves using an aggregate statistic that acts as a graph representaion and a traditional classifier to sort the different frames. The next method involves a graph neural network architecture that utilizes graph convolutional and pooling layers to categorize the different frames. Both methods show to be effective methods for classifying the different classes of mitochondira.

Background

Mitochondria

Mitochondria are double-membrane organelles that act as the powerhouse of the cell because they generate a high amount of Adenosine triphosphate (ATP), an energy-carrying molecule that is essential for many fuctions and processes in living cells, modulate programmed cell death pathways [RJCC19]. One of their critical roles includes shaping the functions of immune cells during infection. Their network structure allows for the dynamic regulation which is necessary to maintain a functional state and allows the mitochondria to be morphologically and functionally independent within cells [KY03]. These morphologies ,fission and fusion, are common events in mitochondria allowing it to continuously change and adapt in response to changes in energy and stress status. Mitochondrial fission, characterized by the cell dispersing and fragmenting over time, allows for damaged organelles to have a quick turnover and fusion allows for the mitochondria to continuously adapt to environmental needs. The fusion of mitochondria is characterized by the mitochondria fusing togheter allowign the mitochondria to merge with other mitochondria that have different defects than itself. Additionally, frequent fusion and fission within the dynamic network is a sign of efficient mitochondrial DNA (mtDNA) complementation as a result of fusing mitochondria which allows for the exchange of genomes [KY03]. These functions are regulated by the frequency of fusion and fusion events. Studies show that the rate of these changes serves as the efficient means of maintaining a good cell environment [LMJH20]. An excess of either function could lead to mitochondrial fragmentation, a sign of cell dysfunction.

Anomalies in a cell's dynamics are very telling of the health of a cell and could be a result of toxic conditions. In recent studies, it has been shown that pathogens attack the host by disturbing the metabolic hub of the cell that is mitochondria. Evidence suggests some pathogens interfere with the mitochondrial network to favor their own replication. Bacteria induce rapid mitochondrial fragmentation by releasing listeriolysin O (LLO) into the mitochondria which causes membrane potential loss and eventually a drop in ATP production [RWH18]. Mitochondria and their dynamics not only help regulate the cell environment but also play a huge role in controlling cell functions during pathogen invasion. Studying the disturbance in these mitochondrial dynamics could help track and detect infections in a quicker manner. Changes in the mitochondria network requires effective detection, and modeling them as a social network and applying graph classification offers a viable solution.

Cell Classification

Advancements to microscopy and deep learning has led way to a new generation of cell and cell morphologies classification techniques. More recently, image based analyses have advanced past single cell classification and are able to allow morphological profiling as seen in [MLTS19]. [MLTS19] examines the advantages and challenges of different machine learning algorithms useful for large-scale label free multi-class cell classification tasks which would be applicable to a diverse set of biological applications ranging from cancer screening to drug identification. The authors propose a single cell classification network that uses a convolutional neural network (CNN) architecture and compare it against traditional methods such as k-nearest neighbors and support vector machines. The CNN architecture proves to be an effective method for human somatic cell types and their morphologies. These morphologies are easier to segment and analyze than spatially diffuse structures like mitochondria.

Transfer learning has also given rise to novel advancements and shows much promise in cell classification tasks [RST+19]. [RST+19] utilizes a hybrid between generative adversarial networks (GANs) and transfer learning dubbed transferring of pertained generating adversarial networks (TOP-GAN) to classify various cancer cells. This approach tackles the main bottleneck of deep learning, small training datasets. To cope with the problem, [RST+19] suggests using a large number of unclassified images from other cell types. This solution is valid only for the context of a few problems. The problem is another label-free multi-class classification problem trying to categorize different types of healthy and unhealthy cancer cells. The context of the problem allows the model to train on a variety of different cells which can then be applied to classify several other types of cells.

Our problem, although having a relatively small data size, does not allow to generalize between different cells. We propose a model that uses only the spatial-temporal aspects of subcellular organelles, in this case the last frames of videos tracking the fusion and fission events, to classify between healthy and unhealthy cells.

Another transfer learning method that deals specifically with classifying organelle morphology is [LPJ+21]. This approach applies CNNs and their advantages of automatic feature engineering and invariance of learning non-linear, input-output mapping to predict morphological abnormalities in plant cells. [LPJ+21] looks at the morphologies of three different subcellular organelles in plant cells, chloroplasts, mitochondria, and peroxisomes to categorize abnormal perturbations. This results in three different types of images for each class with numerous organelles distributed across every image. Nine variants of five different CNN-based models were tested, Inception-v3 [SVI+16], VGG16 [SZ14], ResNet [HZRS16], DenseNet220 [HLvdMW17], and MobileNet-v2 [SHZ+18], all of which proved to be effective methods.

Our problem deals primarily with using mitochondria to categorize anomalies in the cell. Plant cells and their functions vary largely compared to human cells. Most work in cell classification, thus far, deals largely with image data as is and utilizes a CNN or hybrid architecture due to their advantages for analyzing visual imagery. We leverage the principles of graph theory to model the mitochondrial patterns as a social network to study the changing topology of the graphs. Additionally, we look to apply a supervised single-class classification to single frames of mitochondria after a morphological change has occurred.

Graph Learning

Graph machine learning has been drawing increasing attention in recent years due to its versatility and numerous applications especially in biomedical research. Graphs offer a unique way to represent complex systems as set of entities (vertices) and relationships (edges) [ZCH+20]. Graphs are able to capture the relationships between several biological entities including cells, genes, molecules, diseases and drugs. This area of deep learning

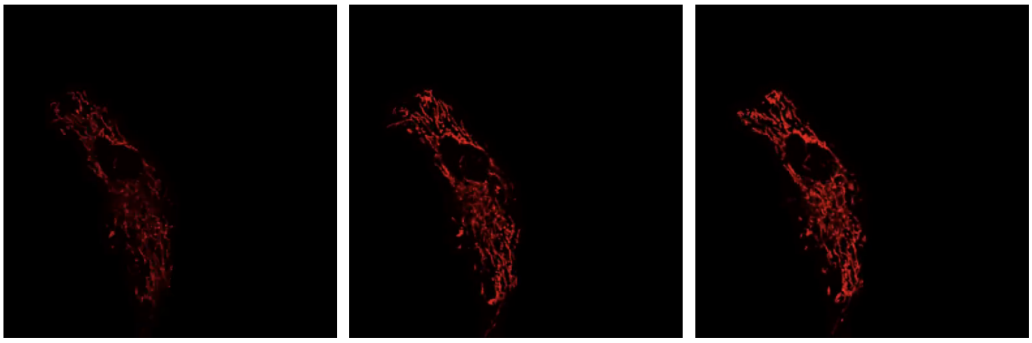

Fig. 1: *The first, middle and last frames on a control cell with no chemical exposure.*

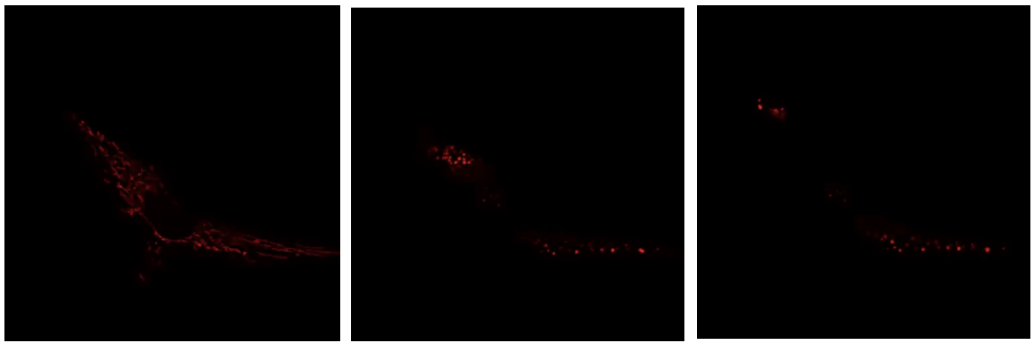

Fig. 2: *The first, middle and last frames on an cell exposed to listeriolysin O (llo). The frames show the resulting fragmentation.*

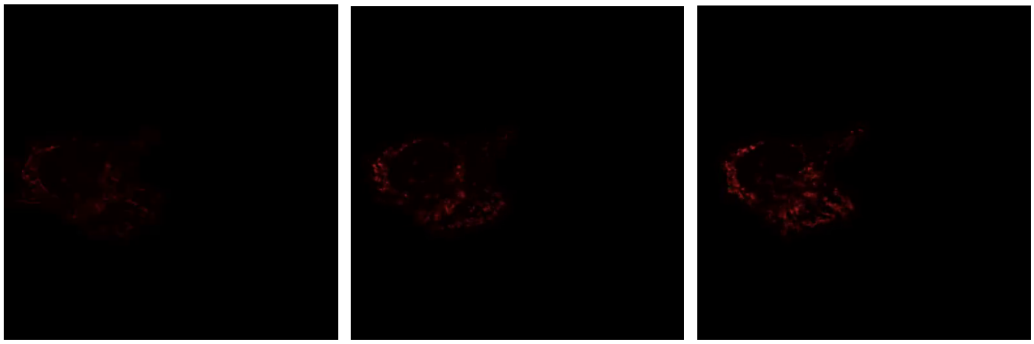

Fig. 3: *The first, middle and last frames on an cell exposed mitochondrial- division inhibitor 1 (mdivi). The frames show the resulting fusion.*

has been showing much promise in modeling the interactions of various cell functions. In our work, we propose classification in a lesser known setting, categorizing the graph as a whole to categorize the different morphologies by analyzing their topologies. Thus, we explored a couple graph neural networks (GNNs), a class of deep learning methods designed to perform inference on graph data [PWB18]. GNNs have proven to be very robust models because they are able to generalize to adapt to dynamic graphs and new unseen graphs. Following the success of word embeddings, node embeddings rose to prominence with DeepWalk [PARS14], an embedding method often referred to as the first graph embedding for representation learning [ZCH+20].

One of our methods does employ a simple embedding method based extracting graph feature information using node feature statistics. Although [HYL17] explains these traditional methods to be limited and sometime inflexible, the method showed favorable results in our experiments. Several new methodologies to produce embeddings followed after DeepWalk but the methods suffer a few drawbacks: node embeddings are computationally inefficient because the number of parameters increased with number of nodes as a result of no shared parameters and the direct embeddings lacked the ability to generalize to a new data. As a means to solve these problems and drawing inspiration to generalize CNNs, GNNs were proposed to aggregate information from the graph structure and better capture the elements and dependencies of the graphs.

There are two main operations at the core of GNNs, convolution and pooling layers. Convolution layers are used to learn a

non-linear transformation of the input graphs perform message passing between the nodes and their neighborhoods. Pooling layers aim to reduce the number of nodes in the graph into a single vector representation and have a similar role to pooling in traditional convolutional neural networks for learning hierarchical representations [GA20].

Because of their general nature, graph neural networks are applicable to three different tasks: node level tasks, link level tasks and graph level tasks. The most applicable task for our problem context is graph level because we attempt to perform classification of graph structures, where each whole graph is assigned a label.

For the context of our problem we utilize graph convolution operations defined by graph convolutional networks in [KW16] and a GCS layer operations used to build graph neural networks with convolutional auto-regressive moving average filters also known as ARMA filters [BGLA21].

Data

Microscopy Imagery

The data consists of a series of live confocal imaging videos that portray the various mitochondrial morphologies in HeLA cells. Figures 1 , 2 and 3 show the raw images of the first, middle and last frames of cells that belong to three different classes. For visualization purposes, the cell was transfected with the DsRed2-Mito-7 protein which gives the mitochondria a red hue. Three different groups of cells with different dynamics were captured: a group experiencing fragmentation from being exposed to toxin listeriolysin O (llo) as seen in figure ref:*fig23*, another group experiencing fusion as a result of being exposed to mitochondrial-division inhibitor 1 (mdivi) as seen in figure 3 and finally a control group that was not exposed to any chemical as seen in figure 1. All the videos were taken using a Nikon A1R confocal microscope. The camera captured 20,000 frames per video with dimensions 512x512 pixels, i.e one image every 10 seconds for the length of the video. All the cells were kept at a temperature of 37 degrees C and 5% CO2 levels for the duration of imaging.

Graph Data

From the 114 videos, we take the last frame and create node features for each single cell video. The dataset we used to train and test our methods contains a node feature matrix and an adjacency matrix for last frames of 114 videos.

The existing OrNet[1] frameworks utilizes Gaussian mixture models (GMMs) to construct the social networks graphs. GMMs were used to determine the spatial regions of the microscope imagery that constructed mitochondrial cluster graphs by iteratively updating the parameters of the underlying mixture distribution until they converged. The parameters of the mixture distributions, post convergence, were used to construct the social network graph [ADATLBR+18], [MHMFRM+20], [FHD+20]. The Gaussian mixture components update over each frame to track the morphologies and the last frames show the social network graph of Gaussians after a series of events. It is for this reason, we use the last frames as the mixture components in the last frame are most indiative of the morphology.

The nodes in the graph correspond to the gaussian mixture components, and the statistics that describe each mixture distribution act as the features. The Gaussian distributions are 2-dimensional, because they model the spatial locations of mitochondrial clusters in the microscopy imagery. Intuitively, the five node features correspond to the location of the Gaussian, the shape of its distribution, and the density of the mitochondrial cluster. Computationally, the location of the gaussian is represented by the pixel coordinates of the center of the distribution, which corresponds to the means of both dimensions; the shape is defined by the variance of each dimension; and the density of the mitochondrial cluster is represented by the number of pixels that are "members" of the mixture component, meaning it is more probable that those pixel belong to the given mixture distribution than any of the others.

After the data preprocessing, there are 114 feature matrices of the shape [N,5] where N is the number of nodes in the mitochondrial cluster and a fully connected adjacency matrix of shape [N,N] that belong to one of three classes: llo which indicates a fusion event, mdivi which indicates a fission event and control, which indicates no abnormal morphology. Both the feature matrix and the adjacency matrix serve as the input to the GNN and there is a target variable associated with each input either 1 or 0 depending on the context of the problem.

Methodology

To contextualize the empirical results, we split the problem up into two different binary classification problems. One problem is to differentiate between the fusion and fission events, i.e categorize between llo and mdivi groups. And the second is to categorize between the fusion event and no abnormal changes i.e, categorize between llo and control and between mdivi and control.

GNN

We trained two different architectures one for each of the two classification problems at hand. One involves a GCN and second is a slightly altered GCN architecture with a trainable skip connection called a GCS layer [BGLA21]. Each of the GCN and GCS layers were followed by a MinCut Pooling layer [BGA19] to get a more refined graph representation after each layer. The models accept a node feature matrix, X, and an adjacency matrix, A; each matrix individually is uninformative to the model but combined they provide the model with enough information about the graph structure. The GCS filter operation is similar to [KW16] with an additional skip connection which has shown to sometimes be more applicable to graph classification. The generally known GCN convolution operation looks like the following,

$$\bar{X}^{t+1} = \sigma(LX^{(t)}W^{(t)})$$

where σ is the non linear activation function, $W^{(t)}$ is the weight matrix at t-th neural network layer and L is the graph Laplacian which can be computed using the normalized grpah adjacency matrix $\hat{A}$ and identiy matrix I. $L = I - \hat{A}$

The GCS operation which has an additional skip connection looks like the following

$$\bar{X}^{t+1} = \sigma(LX^{(t)}W^{(t)} + XV)$$

where σ is the non linear activation function that can be ReLU, sigmoid or hyperbolic tangent (tanh) functions. W and V are trainable parameters. L is the graph Laplacian which can be computed using the normalized grpah adjacency matrix $\hat{A}$ and identiy matrix I. $L = I - \hat{A}$ Each GCS layer is localized in the node space, and it performs a filtering operations between the local

1. https://github.com/quinngroup/ornet

neighboring nodes through the skip connection and the initial node features X [BGLA21].

The graph convolution layer of each model is followed by the MinCut Pooling layer [BGA19]. This method is based on the minCUT optimization problem which finds a cut of the graph that still preserves the topology and representation of the graph. It computes a soft clustering of the input graphs and outputs a reduced node features and adjacency matrix. The dimensions are reduced to the parameter k which is specified when calling the pooling layer. Finally, the last layer of both architectures is a global pooling architecture that pools the graph by computing the sum of the inputs node features. Then the model is through a Dense layer, a fully connected output layer The architectures were trained using Adam optimizer, and L2 penalty loss with weight 1e-3 and 16 hidden units. The GCS layers used a tanh activation function. The MinCut pooling layer is set to output N/2 nodes in the first layer and N/4 at the second layer and N is the average order of the graphs in the dataset. The Dense layer used a sigmoid activation function and we used binary cross entropy for the loss. The models ran for 3000 epochs.

Graph level features using node statistics

This approach deals with finding a good graph representation by using a method similar to bag of nodes. Because the available number of graphs for each class are limited, we create a graph feature by reducing the node features to a vector of statistics. We created four different statistics to act as the graph features: min, max, mean and median. Meaning, for each of the node features, one aggregate statistic (min, max, mean or median) is applied to create a vector of size 5 that would serve as an input for the classifiers. After all the data instances are reduced to a vector, we apply a stratified split using an 80-20 train-test-split. Note, the stratified split preserves the proportions of the classes. This is done before any oversampling technique to ensure that all the samples used for testing are from the original data. Then for the training set we apply the synthetic minority oversampling technique (SMOTE) to oversample the minority classes as a solution to combat the class imbalance. A dataset with imbalanced classes such as the case in this problem could keep a classifier from effectively learning the decision boundary. SMOTE [CBHK02] does not simply duplicate the elements of the minority class but rather synthesizes new instances. This unique oversampling technique selects examples that are close to the original elements in the feature space by drawing a line between two random existing instances and creating a new instance at a point along the line. This method is very effective because the new samples that are created are realistic instances of the minority class and it helps balance the class distributions. We used oversampled graph features as input data for three traditional machine learning algorithms to classify the features into a specific class, k-nearest neighbors, decision tree classifier and random forest classifier.

Experiments and Results

We test the performance of our methods on three different classification tasks: (i) categorize between the last frame images of mitochondria that have been exposed to toxin listeriolysin (class llo) and mitochondria that have been exposed to mitochondrial-division inhibitor 1 (class mdivi), (ii)categorize between the last frames of mitochondria that have been exposed to toxin listeriolysin (class llo) and mitochondria that was exposed to no external stimuli to serve as a control group (class control) and (iii) categorize between mitochondria that have been exposed to mitochondrial- division inhibitor 1 (class mdivi) and mitochondria that was exposed to no external stimuli to serve as a control group (class control). The three classification problems help evaluate all possible differences in the morphologies. Both classification tasks that deal with distinguishing between class llo versus class control and class mdivi versus class control are meant to explore whether our methods can distinguish between anomalous and healthy cells. The classification task that deals with llo and mdivi data investigates whether the methods can distinguish between two different types of anomalies (fusion and fission).

Due to the class imbalance and relatively small size of the dataset, (llo had 54 instances, mdivi had 31 instances and control had 29 instances) we decided to take two different approaches for the methods. One solution was to downsample the llo class which is the majority class to help the GNN methods. We also used this downsampling method for the traditional classifiers to compare the different methodologies effectively. Specifically, this downsampling technique was chosen to keep the model from randomly guessing the llo class for every test instance. Therefore, 19 frames of each of the three classes were used for training and 12 frames were used for testing. The sequence of frames that were in the training and test sets for each run varied as they were randomly subsampled for each time. We used two GNN architectures and three different classifiers with four aggregate stattistics resulting in twelve traditonal methods total.

Alternatively, we utilized an oversampling technique on the input data, which consited of the graph representation vectors, for the traditional classifiers. The input data for the traditional classifiers was first split into training and test sets. Eigty percent of each class was reserved for testing and the remaining twenty perenct for testing. The frames chosen for training and test set for each run were randomly subsampeld for each run. Then sythetic minority oversampling technique (SMOTE) was applied to the data reserved for training to balance the classes. After oversampling, the training set for the Llo-Control classification problem had 44 samples of each class and the test set had 6 control instances and 10 llo instances. The Mdivi-Llo task also had 44 instances of each class in the training set and had a test set consisting of 7 mdivi instances and 10 llo. Lastly, the Mdivi-Control task had 25 instances of each class for training and 6 instances of each respective class for testing. The train-test split was applied prior to oversampling to ensure that only real data points are used for testing. Oversampling was only possible with the data for the traditonal methods as it is not possible to apply an oversampling technique to create entire graphs and their node features. The input data for GNNs is a graph and its node features. Furthermore, the shape of each graph varied based on the instance which would make oversampling difficult and ineffective at producing new data instances.

Both the traditional classifier and GNN methods fully train on the test set and evaluate on the testing set. We measured the number of correctly classified instances of each model and used the accuracy as the main metric to evaluate the performance of our models. Additionally, we include the precision, recall and F-1 scores for each class to show the statistical significance of the results.

Tables 1, 2, 3 contain the results for oversampled data using traditional classifiers. Table 1 shows the results for classifying mdivi and llo data instances using oversampling with SMOTE.

	Accuracy	Precision		Recall		F-1 Score	
	-	Mdivi	LLO	Mdivi	LLO	Mdivi	LLO
Median - Random Forest	0.770	0.908	0.739	0.500	0.959	0.624	0.832
Mean - Random Forest	0.743	0.871	0.718	0.452	0.948	0.574	0.814
Min - Random Forest	**0.812**	0.893	0.792	0.629	0.941	0.721	0.857
Max - Random Forest	0.707	0.824	0.687	0.374	0.939	0.494	0.791
Median - Decision Trees	0.764	0.890	0.737	0.495	0.952	0.613	0.828
Mean - Decision Trees	0.741	0.860	0.718	0.455	0.941	0.572	0.812
Min - Decision Trees	0.781	0.866	0.762	0.565	0.931	0.664	0.835
Max - Decision Trees	0.720	0.825	0.702	0.418	0.932	0.531	0.798
Median - kNN	0.670	0.588	0.752	0.662	0.676	0.615	0.705
Mean - kNN	0.747	0.718	0.778	0.650	0.815	0.669	0.790
Min - kNN	0.702	0.610	0.795	0.725	0.686	0.659	0.732
Max - kNN	0.579	0.479	0.646	0.464	0.659	0.462	0.647

TABLE 1: *Results for Mdivi vs. LLO task using traditional classifiers and SMOTE oversampling technique*

	Accuracy	Precision		Recall		F-1 Score	
	-	Control	LLO	Control	LLO	Control	LLO
Median - Random Forest	0.745	0.835	0.733	0.413	0.944	0.530	0.823
Mean - Random Forest	0.780	0.937	0.752	0.446	0.979	0.581	0.849
Min - Random Forest	0.739	0.819	0.730	0.403	0.941	0.517	0.820
Max - Random Forest	**0.826**	0.927	0.804	0.586	0.970	0.696	0.876
Median - Decision Trees	0.749	0.837	0.737	0.421	0.945	0.536	0.826
Mean - Decision Trees	0.763	0.875	0.746	0.439	0.958	0.559	0.836
Min - Decision Trees	0.721	0.757	0.721	0.393	0.918	0.493	0.805
Max - Decision Trees	0.814	0.923	0.791	0.555	0.969	0.671	0.869
Median - kNN	0.636	0.512	0.714	0.509	0.712	0.500	0.708
Mean - kNN	0.703	0.635	0.739	0.500	0.825	0.545	0.776
Min - kNN	0.634	0.504	0.711	0.493	0.719	0.488	0.710
Max - kNN	0.560	0.391	0.651	0.388	0.664	0.382	0.652

TABLE 2: *Results for Control vs. LLO task using traditional classifiers and SMOTE oversampling technique*

	Accuracy	Precision		Recall		F-1 Score	
	-	Control	Mdivi	Control	Mdivi	Control	Mdivi
Median - Random Forest	**0.781**	0.905	0.731	0.637	0.924	0.731	0.811
Mean - Random Forest	0.750	0.876	0.705	0.595	0.905	0.688	0.786
Min - Random Forest	0.755	0.904	0.704	0.580	0.931	0.685	0.795
Max - Random Forest	0.763	0.890	0.717	0.610	0.916	0.704	0.798
Median - Decision Trees	0.734	0.888	0.683	0.546	0.921	0.653	0.778
Mean - Decision Trees	0.731	0.865	0.686	0.562	0.900	0.659	0.772
Min - Decision Trees	0.719	0.869	0.670	0.524	0.913	0.630	0.767
Max - Decision Trees	0.737	0.870	0.692	0.566	0.908	0.664	0.778
Median - kNN	0.613	0.692	0.591	0.433	0.794	0.512	0.671
Mean - kNN	0.691	0.726	0.677	0.602	0.781	0.648	0.719
Min - kNN	0.576	0.600	0.566	0.444	0.708	0.496	0.622
Max - kNN	0.596	0.590	0.604	0.555	0.637	0.563	0.611

TABLE 3: *Results for Control vs. Mdivi task using traditional classifiers and SMOTE oversampling technique*

For this task, random forest classifer using the min aggregate statistic produced the best results with an accuracy of 0.812. Table 2 shows the results for classifying llo and control data instances using oversampling with SMOTE. Max random forest had the performed in distinguishing control versus llo frames with an accuracy of 0.826. Table 3 shows the results for classifying mdivi and control data instances using oversampling with SMOTE with Median-Random Forest having the highest accuracy at 0.781.

Tables 5, 4, 6 contain the results for of the traditional classifiers and the graph neural network architectures with the downsampled data. Table 5 shows the results for control-llo classification task with Max-Random Forest and GNNs with GCS layers having best accuracy of 0.68 and 0.686 respectively. Table 5 shows the results for mdivi-llo classificaiton. This task had four methods that had the best accuracy, GNN with GCS layers with an accuracy of 0.736 and Mean-Random Forest, Median-Decision trees and Max-kNN all three of which had an accuracy of 0.73. Lastly, table 6 shows the results for Mdivi-control classification. The highest accuracy for this task was Min-Random Forest with an accuracy of 0.619.

Discussion

Overall, both methods have proven to be effective in classifying anomalies in mitochondria. The methods also prove that the node features effectively capture the properties of three different organelle morphologies and graphs are an effective way to represent mitochondria. It is clear from the results that oversampling the data is a good way to train the models well and make better predictions. So, it is worth noting that especially the deep learning models, which are known to be extremely data hungry, could benefit even more so from having more data.

When the data is oversampled, the random forest classifier performs well consitently but the aggregate statistic varies for

each task. It is also interesting to note that the recall metric is disproportionately better for one class in every task. For llo-mdivi and llo-control tasks, this can potentially be attributed to oversampling the minority class and the majority class, which is llo in both cases, having more real data instances.

When the data was downsampled, there was a considerable drop in performance as depicted by tables 5, 4, 6. In this sampling method, the recall scores for the two classes in all three tasks appear to be closer which can again be potentially be attributed to training on all real data. The best metrics varied across each of the tasks. Graph deep learning methods performed well in the control-llo task and mdivi-llo task. Random forest continued to oupefrom most methods except for the mdivi-llo task, in which Max-kNN and Median-decision trees had high accuracies.

Conclusion

Healthy dynamics of subcellular organelles are vital to their metabolic functions. Identifying anomalies in the dynamics is a challenging but important task. In this work, we propose two approaches to classifying different cell morphologies utilizing only the last frames of videos capturing mitochondrial fusion and fission. One method takes the node features and applies a general statistic to make one graph level feature to serve as input for a traditional classifier. Another approach proposes using a graph neural network architecture to perform graph classification that take in a node feature matrix and an adjacency matrix as inputs. We show that both approaches are effective ways to classify between anomalous and regular mitochondria and between two different types of anomalous morphologies. Furthermore, we prove graph neural networks show much promise in classifying and perhaps even tracking the mitochondria and their morphologies.

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.

[BGA19] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Mincut pooling in graph neural networks, 06 2019.

[BGLA21] Filippo Maria Bianchi, Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Graph neural networks with convolutional arma filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP:1–1, 01 2021. doi:10.1109/TPAMI.2021.3054830.

[CBHK02] Nitesh Chawla, Kevin Bowyer, Lawrence Hall, and W. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Intell. Res. (JAIR)*, 16:321–357, 06 2002. doi:10.1613/jair.953.

[DSF13] Giovanni Delogu, Michela Sali, and Giovanni Fadda. The biology of mycobacterium tuberculosis infection. *Mediterranean journal of hematology and infectious diseases*, 5:e2013070, 11 2013. doi:10.4084/MJHID.2013.070.

[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.

[GA20] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral, 06 2020.

[HLvdMW17] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Weinberger. Densely connected convolutional networks. 07 2017. doi:10.1109/CVPR.2017.243.

[HYL17] William Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. 06 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016. doi:10.1109/CVPR.2016.90.

[KW16] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. 09 2016.

[KY03] M Karbowski and R.J. Youle. Karbowski m, youle rj.. dynamics of mitochondrial morphology in healthy cells and during apoptosis. cell death differ 10: 870-880. *Cell death and differentiation*, 10:870–80, 09 2003. doi:10.1038/sj.cdd.4401260.

[LMJH20] Yasmine Liu, Rebecca McIntyre, Georges Janssens, and Riekelt Houtkooper. Mitochondrial fission and fusion: A dynamic role in aging and potential target for age-related disease. *Mechanisms of Ageing and Development*, 186:111212, 02 2020. doi:10.1016/j.mad.2020.111212.

[LPJ+21] Jiying Li, Jinghao Peng, Xiaotong Jiang, Anne Rea, Jiajie Peng, and Jianping Hu. Deeplearnmor: a deep-learning framework for fluorescence image-based classification of organelle morphology. *Plant Physiology*, 05 2021. doi:10.1093/plphys/kiab223.

[MHMFRM+20] Marcus Hill, Mojtaba Fazli, Rachel Mattson, Meekail Zain, Andrew Durden, Allyson T Loy, Barbara Reaves, Abigail Courtney, Frederick D Quinn, S Chakra Chennubhotla, and Shannon P Quinn. Spectral Analysis of Mitochondrial Dynamics: A Graph-Theoretic Approach to Understanding Subcellular Pathology. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 91 – 97, 2020. doi:10.25080/Majora-342d178e-00d.

[MLTS19] Nan Meng, Edmund Y. Lam, Kevin K. Tsia, and Hayden Kwok-Hay So. Large-scale multi-class image-based cell classification with deep learning. *IEEE Journal of Biomedical and Health Informatics*, 23(5):2091–2098, 2019. doi:10.1109/JBHI.2018.2878878.

[OHL+19] Ronald Oei, Guanqun Hou, Fuhai Liu, Jin Zhong, Jiewen Zhang, Zhaoyi An, Luping Xu, and Yujiu Yang. Convolutional neural network for cell classification using microscope images of intracellular actin networks. *PLOS ONE*, 14:e0213626, 03 2019. doi:10.1371/journal.pone.0213626.

[PARS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 03 2014. doi:10.1145/2623330.2623732.

[PWB18] Victor Bapst Alvaro Sanchez-Gonzalez Vinicius Zambaldi Mateusz Malinowski Andrea Tacchetti David Raposo Adam Santoro Ryan Faulkner Caglar Gulcehre Francis Song Andrew Ballard Justin Gilmer George Dahl Ashish Vaswani Kelsey Allen Charles Nash Victoria Langston Chris Dyer Nicolas Heess Daan Wierstra Pushmeet Kohli Matt Botvinick Oriol Vinyals Yujia Li Razvan Pascanu Peter W. Battaglia, Jessica B. Hamrick. Relational inductive biases, deep learning, and graph networks. 2018. doi:10.1145/2623330.2623732.

[RJCC19] Elodie Ramond, Anne Jamet, Mathieu Coureuil, and Alain Charbit. Pivotal role of mitochondria in macrophage response to bacterial pathogens. *Frontiers in Immunology*, 10:2461, 10 2019. doi:10.3389/fimmu.2019.02461.

[RST+19] Moran Rubin, Omer Stein, Nir Turko, Yoav Nygate, Darina Roitshtain, Lidor Karako, Itay Barnea, Raja Giryes, and Natan Shaked. Top-gan: Label-free cancer cell classification using deep learning with a small training set. *Medical Image Analysis*, 57, 06 2019. doi:10.1016/j.media.2019.06.014.

[RWH18] Lateef Reshi, Hao-Ven Wang, and Jiann Hong. *Modulation of Mitochondria During Viral Infections*. 01 2018. doi:10.5772/intechopen.73036.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks forclassification, detection and segmentation. 01 2018.

	Accuracy	Precision		Recall		F-1 Score	
	-	Control	LLO	Control	LLO	Control	LLO
GNN with GCS Layers	0.59	0.58	0.6	0.59	0.59	0.58	0.59
GNN with GCS Layers	**0.686**	0.5	0.83	0.75	0.62	0.6	0.71
Median - Random Forest	0.59	0.6	0.58	0.55	0.64	0.57	0.61
Mean - Random Forest	0.45	0.44	0.46	0.36	0.55	0.4	0.5
Min - Random Forest	0.41	0.38	0.43	0.27	0.55	0.32	0.48
Max - Random Forest	**0.68**	0.7	0.67	0.64	0.73	0.67	0.7
Median - Decision Trees	0.59	0.6	0.58	0.55	0.64	0.57	0.61
Mean - Decision Trees	0.55	0.54	0.56	0.64	0.45	0.58	0.5
Min - Decision Trees	0.5	0.5	0.5	0.36	0.64	0.42	0.56
Max - Decision Trees	0.64	0.71	0.6	0.45	0.82	0.56	0.69
Median - kNN	0.55	0.57	0.53	0.36	0.73	0.44	0.62
Mean - kNN	0.41	0.25	0.44	0.09	0.73	0.13	0.55
Min - kNN	0.41	0.33	0.44	0.18	0.64	0.24	0.52
Max - kNN	0.41	0.38	0.43	0.27	0.55	0.32	0.48

TABLE 4: *Results for Control vs. LLO task using traditional classifiers and GNNs. The data was undersampled meaning the training set had 19 instances of each class and the test set had 11 instances of each class.*

	Accuracy	Precision		Recall		F-1 Score	
	-	Mdivi	LLO	Mdivi	LLO	Mdivi	LLO
GNN with GCS Layers	**0.736**	0.75	0.67	0.69	0.73	0.72	0.7
GNN with GCS Layers	0.58	0.58	0.58	0.58	0.58	0.58	0.58
Median - Random Forest	0.55	0.55	0.55	0.55	0.55	0.55	0.55
Mean - Random Forest	**0.73**	0.73	0.73	0.73	0.73	0.73	0.73
Min - Random Forest	0.55	0.55	0.55	0.55	0.55	0.55	0.55
Max - Random Forest	0.45	0.45	0.45	0.45	0.45	0.45	0.45
Median - Decision Trees	**0.73**	0.78	0.69	0.64	0.82	0.7	0.75
Mean - Decision Trees	0.64	0.71	0.6	0.45	0.82	0.56	0.69
Min - Decision Trees	0.55	0.56	0.54	0.45	0.64	0.5	0.58
Max - Decision Trees	0.55	0.53	0.57	0.73	0.36	0.62	0.44
Median - kNN	0.5	0.5	0.5	0.36	0.64	0.42	0.56
Mean - kNN	0.5	0.5	0.5	0.64	0.36	0.56	0.42
Min - kNN	0.59	0.56	0.75	0.91	0.27	0.69	0.4
Max - kNN	**0.73**	0.73	0.73	0.73	0.73	0.73	0.73

TABLE 5: *Results for Mdivi vs. LLO task using traditional classifiers and GNNs. The data was undersampled meaning the training set had 19 instances of each class and the test set had 11 instances of each class.*

	Accuracy	Precision		Recall		F-1 Score	
	-	Control	Mdivi	Control	Mdivi	Control	Mdivi
GNN with GCS Layers	0.619	0.6	0.64	0.64	0.6	0.64	0.6
GNN with GCN Layers	0.57	0.6	0.55	0.55	0.6	0.57	0.57
Median - Random Forest	0.55	0.56	0.54	0.45	0.64	0.5	0.58
Mean - Random Forest	0.64	0.67	0.62	0.55	0.73	0.6	0.67
Min - Random Forest	**0.69**	0.69	0.68	0.80	0.56	0.73	0.62
Max - Random Forest	0.55	0.57	0.53	0.36	0.73	0.44	0.62
Median - Decision Trees	0.55	0.56	0.54	0.45	0.64	0.5	0.58
Mean - Decision Trees	0.64	0.67	0.62	0.55	0.73	0.6	0.67
Min - Decision Trees	0.55	0.55	0.55	0.55	0.55	0.55	0.55
Max - Decision Trees	0.59	0.62	0.57	0.45	0.73	0.53	0.64
Median - kNN	0.5	0.5	0.5	0.45	0.55	0.48	0.52
Mean - kNN	0.57	0.7	0.59	0.34	0.66	0.52	0.62
Min - kNN	0.64	0.67	0.62	0.55	0.73	0.6	0.67
Max - kNN	0.55	0.57	0.53	0.36	0.73	0.44	0.62

TABLE 6: *Results for Mdivi vs. Control task using traditional classifiers and GNNs. The data was undersampled meaning the training set had 19 instances of each class and the test set had 11 instances of each class.*

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and ZB Wojna. Rethinking the inception architecture for computer vision. 06 2016. doi:10.1109/CVPR.2016.308.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[YRS19] Kai Yao, Nash Rochman, and Sean Sun. Cell type classification and unsupervised morphological phenotyping from low-resolution images using deep learning. *Scientific Reports*, 9:1–13, 09 2019. doi:10.1038/s41598-019-50010-9.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 01 2020. doi:10.1016/j.aiopen.2021.01.001.

Monitoring Scientific Python Usage on a Supercomputer

Rollin Thomas‡*, Laurie Stephey‡*, Annette Greiner‡, Brandon Cook‡

Abstract—In 2021, more than 30% of users at the National Energy Research Scientific Computing Center (NERSC) used Python on the Cori supercomputer. To determine this we have developed and open-sourced a simple, minimally invasive monitoring framework that leverages standard Python features to capture Python imports and other job data via a package called "Customs". To analyze the data we collect via Customs, we have developed a Jupyter-based analysis framework designed to be interactive, shareable, extensible, and publishable via a dashboard. Our stack includes Papermill to execute parameterized notebooks, Dask-cuDF for multi-GPU processing, and Voila to render our notebooks as web-based dashboards. We report preliminary findings from Customs data collection and analysis. This work demonstrates that our monitoring framework can capture insightful and actionable data including top Python libraries, preferred user software stacks, and correlated libraries, leading to a better understanding of user behavior and affording us opportunity to make increasingly data-driven decisions regarding Python at NERSC.

Index Terms—HPC, Python monitoring, GPUs, dashboards, parallel, Jupyter

Introduction

The National Energy Research Scientific Computing Center (NERSC) is the primary scientific computing facility for the US Department of Energy's Office of Science. Some 8,000 scientists use NERSC to perform basic, non-classified research in predicting novel materials, modeling the Earth's climate, understanding the evolution of the Universe, analyzing experimental particle physics data, investigating protein structure, and more [OA20]. NERSC procures and operates supercomputers and massive storage systems under a strategy of balanced, timely introduction of new hardware and software technologies to benefit the broadest possible portion of this workload. While procuring new systems or supporting users of existing ones, NERSC relies on detailed analysis of its workload to help inform strategy.

Workload analysis is the process of collecting and marshaling data to build a picture of how applications and users really interact with and utilize systems. It is one part of a procurement strategy that also includes surveys of user and application requirements, emerging computer science research, developer or vendor roadmaps, and technology trends. Understanding our workload helps us engage in an informed way with stakeholders like funding agencies, vendors, developers, users, standards bodies, and other high-performance computing (HPC) centers. In particular, workload analysis informs non-recurring engineering contracts where NERSC partners with external software developers to address gaps in system programming environments. Actively monitoring the workload also enables us to identify suboptimal or potentially problematic user practices and address them through direct intervention, improving documentation, or simply making it easier for users to use the software better. Measuring the relative frequency of use of different software components can help us streamline delivery, retiring less-utilized packages, and promoting timely migration to newer versions. Detecting and analyzing trends in user behavior with software over time also helps us anticipate user needs and prepare accordingly. Comprehensive, quantitative workload analysis is a critical tool in keeping NERSC a productive supercomputer center for science.

With Python assuming a key role in scientific computing, it makes sense to apply workload analysis to Python in production settings like NERSC's Cray XC-40 supercomputer, Cori. Once viewed in HPC circles as merely a cleaner alternative to Perl or Shell scripting, Python has evolved into a robust platform for orchestrating simulations, running complex data processing pipelines, managing artificial intelligence workflows, visualizing massive data sets, and more. Adapting workload analysis practices to scientific Python gives its community the same data-driven leverage that other language communities at NERSC already enjoy.

This article documents NERSC's Python workload analysis efforts, part of an initiative called Monitoring of Data Services (MODS) [MODS], and what we have learned during this process. In the next section, we provide an overview of related work including existing tools for workload data collection, management, and analysis. In Methods, we describe an approach to Python-centric workload analysis that uses built-in Python features to capture usage data, and a Jupyter notebook-based workflow for exploring the data set and communicating what we discover. Our Results include high-level statements about what Python packages are used most often and at what scale on Cori, but also some interesting deeper dives into use of certain specific packages along with a few surprises. In the Discussion, we examine the implications of our results, share the strengths and weaknesses of our workflow and our lessons learned, and outline plans for improving the analysis to better fill out the picture of Python at NERSC. The Conclusion suggests some areas for future work.

* *Corresponding author: rcthomas@lbl.gov, lastephey@lbl.gov*
‡ *National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 1 Cyclotron Road MS59-4010A, Berkeley, California, 94720*
* *Corresponding author: rcthomas@lbl.gov, lastephey@lbl.gov*

Related Work

The simplest approach used to get a sense of what applications run on a supercomputer is to scan submitted batch job scripts for executable names. In the case of Python applications, this is problematic since users often invoke Python scripts directly instead of as an argument to the `python` executable. This method also provides only a crude count of Python invocations and gives little insight into deeper questions about specific Python packages.

Software environment modules [Fur91] are a common way for HPC centers to deliver software to users. Environment modules operate primarily by setting, modifying, or deleting environment variables upon invocation of a module command (e.g. `module load`, `module swap`, or `module unload`) This provides an entrypoint for software usage monitoring. Staff can inject code into a module load operation to record the name of the module being loaded, its version, and other information about the user's environment. Lmod, a newer implementation of environment modules [Mcl11], provides documentation on how to configure it to use syslog and MySQL to collect module loads through a hook function. Counting module loads as a way to track Python usage has the virtue of simplicity. However, users often include module load commands in their shell resource files (e.g., *.bashrc*), meaning that user login or shell invocation may trigger a detection even if the user never actually uses the trigger module. Furthermore, capturing information at the package level using module load counts would also require that individual Python packages be installed as separate environment modules. Module load counts also miss Python usage from user-installed Python environments or in containers.

Tools like ALTD [Fah10] and XALT [Agr14] are commonly used in HPC contexts to track library usage in compiled applications. The approach is to introduce wrappers that intercept the linker and batch job launcher (e.g. `srun` in the case of Slurm used at NERSC). The linker wrapper can inject metadata into the executable header, take a census of libraries being linked in, and forward that information to a file or database for subsequent analysis. Information stored in the header at link time is dumped and forwarded later by the job launch wrapper. On systems where all user applications are linked and launched with instrumented wrappers, this approach yields a great deal of actionable information to HPC center staff. However, popular Python distributions such as Anaconda Python arrive on systems fully built, and can be installed by users without assistance from center staff. Later versions of XALT can address this through an `LD_PRELOAD` setting. This enables XALT to identify compiled extensions that are imported in Python programs using a non-instrumented Python, but pure Python libraries currently are not detected. XALT is an active project so this may be addressed in a future release.

[Mac17] describes an approach to monitoring Python package use on Blue Waters using only built-in Python features: `sitecustomize` and `atexit`. During normal Python interpreter start-up, an attempt is made to import a module named `sitecustomize` that is intended to perform site-specific customizations. In this case, the injected code registers an exit handler through the `atexit` standard library module. This exit handler inspects `sys.modules`, a dictionary that normally describes all packages imported in the course of execution. On Blue Waters, `sitecustomize` was installed into the Python distribution installed and maintained by staff. Collected information was stored to plain text log files. An advantage of this approach is that

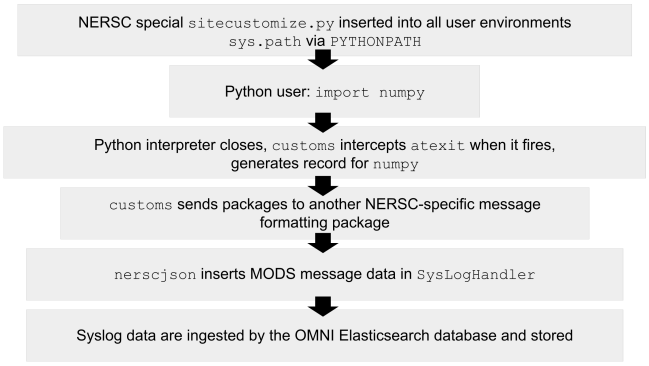

Fig. 1: *NERSC infrastructure for capturing Python usage data.*

`sitecustomize` failures are nonfatal, and placing the import reporting step into an exit hook (as opposed to instrumenting the import mechanism) means that it minimizes interference with normal operation of the host application. The major limitation of this strategy is that abnormal process terminations prevent the Python interpreter from proceeding through its normal exit sequence and package import data are not recorded.

Of course, much more information may be available through tools based on the extended Berkeley Packet Filter and the BPF compiler collection, similar to the `pythoncalls` utility that summarizes method calls in a running application. While eBPF overheads are very small, this approach requires special compilation flags for Python and libraries. Effort would be needed to make the monitoring more transparent to users and to marshal the generated data for subsequent analysis. This could be an interesting and fruitful approach to consider. Obviously, solutions that can overly impact application reliability or place an undue burden on system administrators and operations staff should be avoided. The fullest picture we currently can obtain comes from a combination of non-intrusive tooling and follow-up with users, using the story we can put together from the data we gather as a starting point for conversation.

Methods

Users have a number of options when it comes to how they use Python at NERSC. NERSC provides a "default" Python to its users through a software environment module, based on the Anaconda Python distribution with modifications. Users may load this module, initialize the Conda tool, and create their own custom Conda environments. Projects or collaborations may provide their users with shared Python environments, often as a Conda environment or as an independent installation altogether (e.g. using the Miniconda installer and building up). Cray provides a basic "Cray Python" module containing a few core scientific Python packages linked against Cray MPICH and LibSci libraries. Python packages are also installed by staff or users via Spack [Gam15], an HPC package manager. NERSC also provides Shifter [Jac16], a container runtime that enables users to run custom Docker containers that can contain Python built however the author desires. With a properly defined kernel-spec file, a user is able to use a Python environment based on any of the above options as a kernel in NERSC's Jupyter service. The goal is to gather data for workload analysis across all of these options.

Monitoring all of the above can be done quite easily by using the strategy outlined in [Mac17] with certain changes. Fig. 1 illustrates the infrastructure we have configured. As in [Mac17] a `sitecustomize` that registers the `atexit` handler is installed in a directory included into all users' Python `sys.path`. The `sitecustomize` module is installed directly on each compute node and not served over network, in order to avoid exacerbating poor performance of Python start-up at scale. We accomplish this by installing it and any associated Python modules into the node system images themselves, and configuring default user environments to include a `PYTHONPATH` setting that injects `sitecustomize` into `sys.path`. Shifter containers include the monitoring packages from the system image via runtime volume mount. Users can opt out of monitoring simply by unsetting or overwriting `PYTHONPATH`. We took the approach of provisioning a system-wide `PYTHONPATH` because we cast a much wider collection net (opt-out) than if we depend on users to install `sitecustomize` (opt-in). This also gives us a centrally managed source of truth for what is monitored at any given time.

Customs: Inspect and Report Packages

To organize `sitecustomize` logic we have created a Python package we call Customs, since it is for inspecting and reporting on Python package imports of particular interest. Customs can be understood in terms of three simple concepts. A **Check** is a simple object that represents a Python package by its name and a callable that is used to verify that the package (or even a specific module within a package) is present in a given dictionary. In production this dictionary should be `sys.modules` but during testing it can be a mock `sys.modules` dictionary. The **Inspector** is a container of Check objects, and is responsible for applying each Check to `sys.modules` (or mock) and returning the names of packages that are detected. Finally, the **Reporter** is an abstract class that takes some action given a list of detected package names. The Reporter action should be to record or transmit the list of detected packages, but exactly how this is done depends on implementation. Customs includes a few reference Reporter implementations and an example of a custom Customs Reporter.

Customs provides an entry point to use in `sitecustomize`, the function `register_exit_hook`. This function takes two arguments. The first is a list of strings or (string, callable) tuples that are converted into Checks. The second argument is the type of Reporter to be used. The exit hook can be registered multiple times with different package specification lists or Reporters if desired.

The intended workflow is that a staff member creates a list of package specifications they want to check for, selects or implements an appropriate Reporter, and passes these two objects to `register_exit_hook` within `sitecustomize.py`. Installing `sitecustomize` to system images generally involves packaging the software as an RPM to be installed into node system images and deployed by system administrators. When a user invokes Python, the exit hook will be registered using the `atexit` standard library module, the application proceeds as normal, and then at normal shutdown `sys.modules` is inspected and detected packages of interest are reported.

Message Logging and Storage

NERSC has developed a lightweight abstraction layer for message logging called nerscjson. It is a simple Python package that consumes JSON messages and forwards them to an appropriate transport layer that connects to NERSC's Operations Monitoring and Notification Infrastructure (OMNI) [Bau19]. Currently this is done with Python's standard `SysLogHandler` from the logging library, modified to format time to satisfy RFC 3339. Downstream from these transport layers, a message key is used to identify the incoming messages, their JSON payloads are extracted, and then forwarded to the appropriate Elasticsearch index. The Customs Reporter used on Cori simply uses nerscjson.

Field	Description
`executable`	Path to Python executable used by this process
`is_compute`	True if the process ran on a compute node
`is_shifter`	True if the process ran in a Shifter container
`is_staff`	True if the user is a member of NERSC staff
`job_id`	Slurm job ID
`main`	Path to application, if any
`num_nodes`	Number of nodes in the job
`qos`	Batch queue of the job
`repo`	Batch job charge account
`subsystem`	System partition or cluster
`system`	System name
`username`	User handle

TABLE 1: *Additional monitoring metadata*

On Cori compute nodes, we use the Cray Lightweight Log Manager (LLM), configured to accept RFC 5424 protocol messages on service nodes. A random service node is chosen as the recipient in order to balance load. On other nodes besides compute nodes, such as login nodes or nodes running user-facing services, rsyslog is used for message transport. This abstraction layer allows us to maintain a stable interface for logging while using an appropriately scalable transport layer for the system. For instance, future systems will rely on Apache Kafka or the Lightweight Distributed Metrics Service [Age14].

Cori has 10,000 compute nodes running jobs at very high utilization, 24 hours a day for more than 340 days in a typical year. The volume of messages arriving from Python processes completing could be quite high, so we have taken a cautious approach of monitoring a list of about 50 Python packages instead of reporting the entire contents of each process's `sys.modules`. This introduces a potential source of bias that we return to in the Discussion, but we note here that Python 3.10 will include `sys.stdlib_module_names`, a frozenset of strings containing the names of standard library modules, that could be used in addition to `sys.builtin_module_names` to remove standard library and built-in modules from `sys.modules` easily. Ultimately we plan to capture all imports excluding standard and built-in packages, except for ones we consider particularly relevant to scientific Python workflows like `multiprocessing`.

To reduce excessive duplication of messages from MPI-parallel Python applications, we prevent reporting from processes with nonzero MPI rank or `SLURM_PROCID`. Other parallel applications using e.g. `multiprocessing` are harder to deduplicate. This moves deduplication downstream to the analysis phase. The key is to carry along enough additional information to enable the kinds of deduplication needed (e.g., by user, by job, by node, etc). Table 1 contains a partial list of metadata captured and forwarded along with package names and versions.

Fields that only make sense in a batch job context are set to a default (`num_nodes: 1`) or left empty (`repo: ""`). Basic job

quantities like node count help capture the most salient features of jobs being monitored. Downstream joins with other OMNI indexes or other databases containing Slurm job data (via `job_id`), identity (`username`), or banking (`repo`) enables broader insights.

In principle it is possible that messages may be dropped along the way to OMNI, since we are using UDP for transport. To control for this source of error, we submit scheduled "canary jobs" a few dozen times a day that run a Python script that imports libraries listed in `sitecustomize` and then exits normally. Matching up those job submissions with entries in Elastic enables us to quantify the message failure rate. Canary jobs began running in October of 2020 and from that time until now (May 2021), perhaps surprisingly, we actually have observed no message delivery failures.

Prototyping, Production, and Publication

OMNI has a Kibana visualization interface that NERSC staff use to visualize Elasticsearch-indexed data collected from NERSC systems, including data collected for MODS. The MODS team uses Kibana for creating plots of usage data, organizing these into attractive dashboard displays that communicate MODS high-level metrics. Kibana is very effective at providing a general picture of user behavior with the NERSC data stack, but the MODS team wanted deeper insights from the data and obtaining these through Kibana presented some difficulty, especially due to the complexities of deduplication we discussed in the previous section. Given that the MODS team is fluent in Python, and that NERSC provides users (including staff) with a productive Python ecosystem for data analytics, using Python tools for understanding the data was a natural choice. Using the same environment and tools that users have access to provides us a way to test how well those tools actually work.

Our first requirement was the ability to explore MODS Python data interactively. However, we also wanted to be able to record that process, document it, share it, and enable others to re-run or re-create the results. Jupyter Notebooks specifically target this problem, and NERSC already runs a user-facing JupyterHub service that enables access to Cori. Members of the MODS team can manage notebooks in a Gitlab instance run by NERSC, or share them with one another (and from Gitlab) using an NBViewer service running alongside NERSC's JupyterHub.

Iterative prototyping of data analysis pipelines often starts with testing hypotheses or algorithms against a small subset of the data and then scaling that analysis up to the entire data set. GPU-based tools with Python interfaces for filtering, analyzing, and distilling data can accelerate this scale-up using generally fewer compute nodes than with CPU-based tools. The entire MODS Python data set is currently about 260 GB in size, and while this could fit into one of Cori's CPU-based large-memory nodes, the processing power available there is insufficient to make interactive analysis feasible. With only CPUs, the main recourse is to scale out to more nodes and distribute the data. This is certainly possible, but being able to interact with the entire data set using a few GPUs, far fewer processes, and without inter-node communication is compelling.

To do interactive analysis, prototyping, or data exploration we use Dask-cudf and cuDF, typically using 4 NVIDIA Volta V100 GPUs coordinated by a Dask-CUDA cluster. The Jupyter notebook itself is started from NERSC's JupyterHub using BatchSpawner (i.e., the Hub submits a batch job to run the notebook on the GPU cluster). The input data, in compressed Parquet format, are read using Dask-cuDF directly into GPU memory. These data are

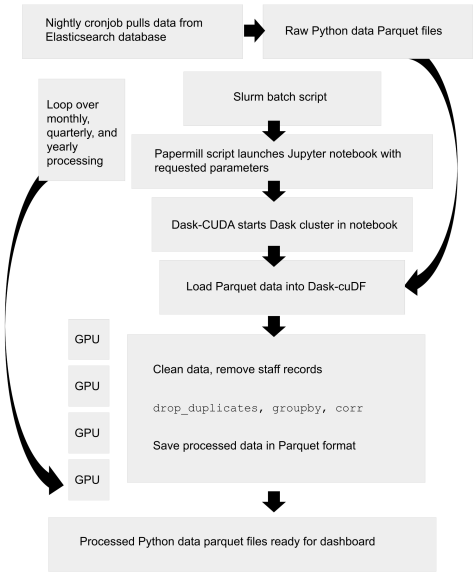

Fig. 2: *Workflow for processing and analyzing Python usage data.*

periodically gathered from OMNI using the Python Elasticsearch API and converted to Parquet. Reduced data products are stored in new Parquet files, again using direct GPU I/O.

As prototype analysis code in notebooks evolves into something resembling a production analysis pipeline, data scientists face the choice of whether to convert their notebooks into scripts or try to stretch their notebook to serve as a production tool. The latter approach has the appeal that production notebooks can be re-run interactively when needed with all the familiar Jupyter notebook benefits. We decided to experiment with using Papermill to parameterize notebook execution over months, quarters, and years of data and submit these notebooks as batch jobs. In each Jupyter notebook, a Dask-CUDA cluster is spun up and then shutdown at the end for memory/worker cleanup. Processing all data for all permutations currently takes about 2 hours on 4 V100 GPUs on the Cori GPU cluster. Fig. 2 illustrates the workflow.

Members of the MODS team can share Jupyter notebooks with one another, but this format may not make for the best way to present data to other stakeholders, in particular center management, DOE program managers, vendors, or users. Voilà is a tool that uses a Jupyter notebook to power a stand-alone, interactive dashboard-style web application. We decided to experiment with Voilà for this project to evaluate best practices for its use at NERSC. To run our dashboards we use NERSC's Docker container-as-a-service platform, called Spin, where staff and users can run persistent web services. Spin is external to NERSC's HPC resources and has no nodes with GPUs, but mounts the NERSC Global Filesystem.

Creating a notebook using a GPU cluster and then using the same notebook to power a dashboard running on a system without GPUs presents a few challenges. We found ourselves adopting a pattern where the first part of the notebook used a Dask cluster and GPU-enabled tools for processing the data, and the second part of the notebook used reduced data using CPUs to power the dashboard visualizations. We used cell metadata tags to direct Voilà to simply skip the first set of cells and pick up dashboard rendering with the reduced data. This process was a little clumsy, and we found it easy to make the mistake of adding a cell and then

forgetting to update its metadata. Easier ways of managing cell metadata tags would improve this process. Another side-effect of this approach is that packages may appear to be imported multiple times in a notebook.

We found that even reduced data sets could be large enough to make loading a Voilà dashboard slow, but we found ways to hide this by lazily loading the data. Using Pandas DataFrames to prepare even reduced data sets for rendering, especially histograms, resulted in substantial latency when interacting with the dashboard. Vaex [vaex] provided for a more responsive user experience, owing to multi-threaded CPU parallelism. We did use some of Vaex's native plotting functionality (in particular `viz.histogram`), but we primarily used Seaborn for plotting with Vaex objects "underneath" which we found to be a fast and friendly way to generate appealing visualizations. Sometimes Matplotlib was used when Seaborn could not meet our needs (to create a stacked barplot, for example).

Finally, we note that the Python environment used for both data exploration and reduction on the GPU cluster, and for running the Voilà dashboard in Spin, is managed using a single Docker image (Shifter runtime on GPU, Kubernetes in Spin).

Results

Our data collection framework yields a rich data set to examine and our workflow enables us to interactively explore the data and translate the results of our exploration into dashboards for monitoring Python. Results presented come from data collected between January and May 2021. Unless otherwise noted, all results exclude Python usage by members of NERSC staff (`is_staff==False`) and include only results collected from batch jobs (`is_compute==True`). All figures are extracted from the Jupyter notebook/Voilà dashboard.

During the period of observation there were 2448 users running jobs that used Python on Cori, equivalent to just over 30% of all NERSC users. 84% of jobs using Python ran on Cori's Haswell-based partition, 14% used Cori-KNL, and 2% used Cori's GPU cluster. 63% of Python users use the NERSC-provided Python module directly (including on login nodes and Jupyter nodes) but only 5% of jobs using Python use the module: Most use a user-built Python environment, namely Conda environments. Anaconda Python provides scientific Python libraries linked against the Intel Math Kernel Library (MKL), but we observe that only about 17% of MKL-eligible jobs (ones using NumPy, SciPy, NumExpr, or scikit-learn) are using MKL. We consider this finding in more detail in Discussion.

Fig. 3 displays the top 20 Python packages in use determined from unique user imports (i.e. how many users ever use a given package) across the system, including login nodes and Jupyter nodes. These top libraries are similar to previous observations reported from Blue Waters and TACC [Mcl11], [Eva15], but the relative prominence of `multiprocessing` is striking. We also note that Joblib, a package for lightweight pipelining and easy parallelism, ranks higher than both mpi4py and Dask.

The relatively low rankings for TensorFlow and PyTorch are probably due to the current lack of GPU resources, as Cori provides access to only 18 GPU nodes mainly for application readiness activities in support of Perlmutter, the next (GPU-based) system being deployed. Additionally, some users that are training deep learning models submit a chain of jobs that may not be expected to finish within the requested walltime; the result is

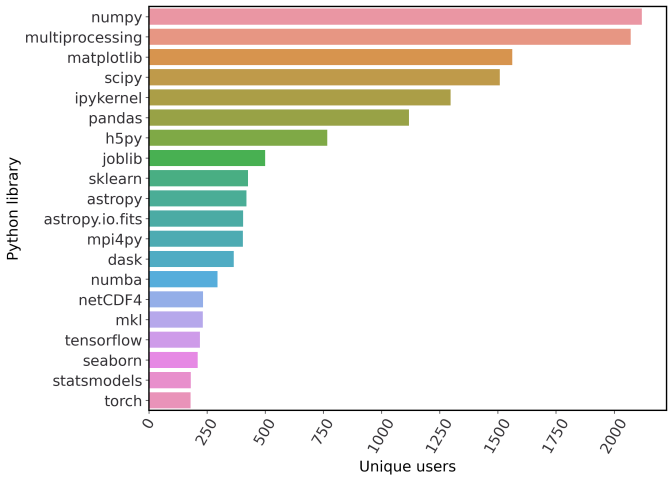

Fig. 3: *Top 20 tracked Python libraries at NERSC, deduplicated by user, across our system.*

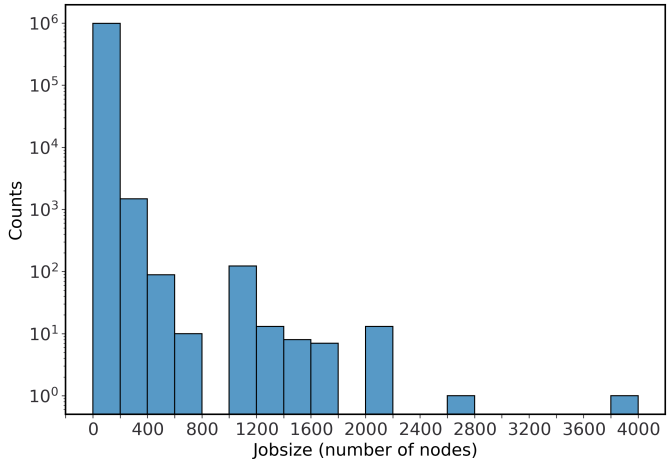

Fig. 4: *Distribution of job size for batch jobs that use Python.*

that the job may end before Customs can capture data from the `atexit`, resulting in under-reporting.

Fig. 4 shows the distribution of job size (node count) for jobs that invoked Python and imported one or more of the packages we monitor. Most of these jobs are small, but the distribution tracks the overall distribution of job size at NERSC.

Breaking down the Python workload further, Fig. 5 contains a 2D histogram of Python package counts as a function of job size. Package popularity in this figure has a different meaning than in Fig. 3: The data are deduplicated by `job_id` and package name to account for jobs where users invoke the same executable repeatedly or invoke multiple applications using the same libraries. The marginal axes summarize the total package counts and total job size counts as a function of `job_id`. Most Python libraries we track do not appear to use more than 200 nodes. Perhaps predictably, `mpi4py` and NumPy are observed at the largest node counts. Dask jobs are observed at 500 nodes and fewer, so it appears that Dask is not being used to scale as large as `mpi4py` is. Workflow managers FireWorks [Jai15] and Parsl [Bab19] are observed scaling to 1000 nodes. PyTorch (`torch`) appears at larger scales than TensorFlow and Keras, which suggests users may find it easier to scale PyTorch on Cori.

While it is obvious that packages that depend on or are

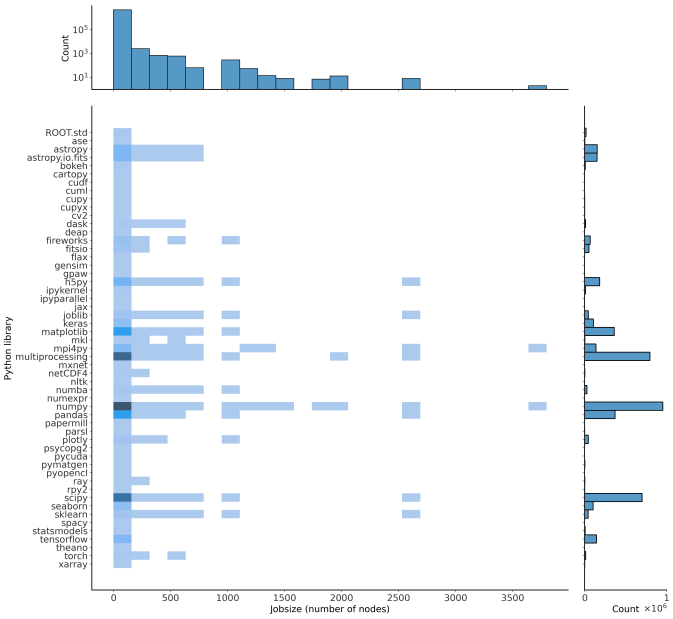

Fig. 5: *2D histogram of Python package counts versus job size. The marginal x-axis (right) shows the total package counts. The marginal y-axis (top) shows the total job counts displayed on a log scale. Here we measure number of unique packages used within a job rather than number of jobs, so these data are not directly comparable to Fig. 3 nor to Fig. 4.*

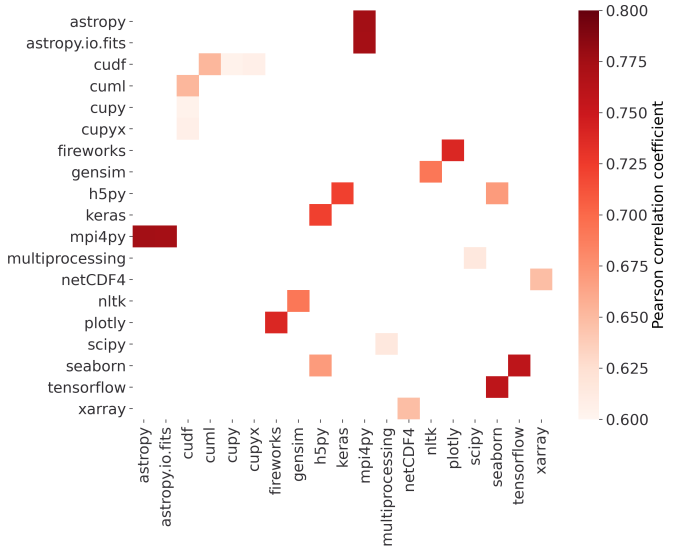

Fig. 6: *Pearson correlation coefficients for tracked Python libraries within the same job. Libraries were only counted once per job. Here we display correlation coefficient values between 0.6 and 0.8 in an effort to highlight a regime in which packages have a strong relationship but no explicit dependencies.*

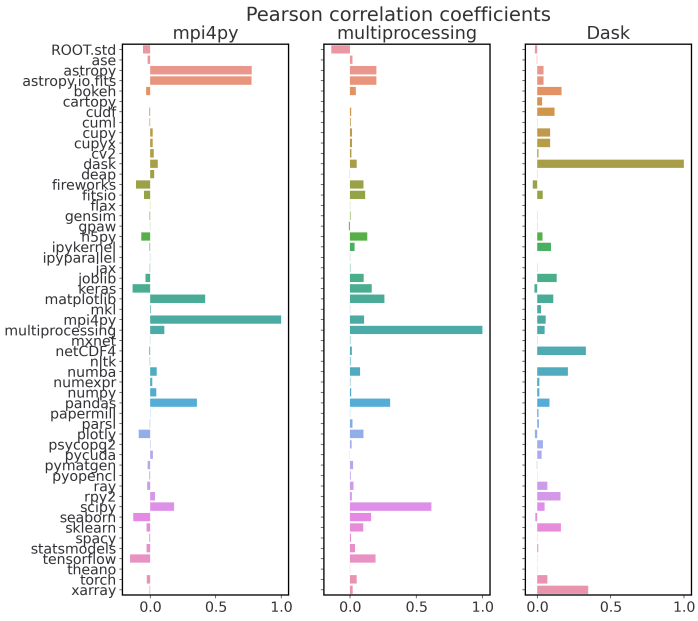

Fig. 7: *Pearson correlation coefficient values for* `mpi4py` *(left),* `multiprocessing` *(center), and Dask (right), with all other Python libraries we currently track.*

dependencies of other packages will be correlated within jobs, it is still interesting to examine the co-occurrence of certain packages within jobs. A simple way of looking at this is to determine Pearson correlation coefficients for each tracked library with all others, assigning a 1 to jobs in which a certain package was used and 0 otherwise. Fig. 6 shows an example package correlation heatmap. The heatmap includes only package correlations above 0.6 to omit less interesting relationships and less than 0.8 as a simple way to filter out interdependencies. Notable relationships between non-dependent packages include `mpi4py` and AstroPy, Seaborn and TensorFlow, FireWorks and Plotly.

We used this correlation information as a starting point for examining package use alongside `mpi4py`, `multiprocessing`, and Dask, all of which we are especially interested in because they enable parallelism within batch jobs. We omit Joblib, noting that a number of packages depend on Joblib and Joblib itself uses `multiprocessing`. Fig. 7 presents the correlations of each of these packages with all other tracked packages.

The strongest correlations observed for `mpi4py` (Fig. 7, left) is the domain-specific package AstroPy and its submodule `astropy.io.fits`. This suggests that users of AstroPy have been able to scale associated applications using `mpi4py` and that AstroPy developers may want to consider engaging with `mpi4py` users regarding their experiences. Following up with users generally reveals that using `mpi4py` for "embarrassingly parallel" calculations is very common: "My go-to approach is to broadcast data using `mpi4py`, split up input hyperparameters/settings/etc. across ranks, have each rank perform some number of computations, and then gather all the results (which are almost always NumPy arrays) using `mpi4py`." Very few users report more intricate communication patterns.

Next we consider `multiprocessing`. The `conda` tool uses `multiprocessing` but even after filtering out those cases, it remains one of the most popular Python libraries in use on Cori. In Fig. 7 (center), we do not see any particularly strong

relationships as we did with `mpi4py`. The primary correlation visible here is with SciPy, which has some built-in support for inter-operating with `multiprocessing`, for instance through `scipy.optimize`. To learn more we followed up with several of the top `multiprocessing` users. One reported: "I'm using and testing many bioinformatics Python-based packages, some of them probably using Python `multiprocessing`. But I'm not specifically writing myself scripts with `multiprocessing`." Another reported: "The calculations are executing a workflow for computing the binding energies of ligands in metal complexes. Since each job is independent, `multiprocessing` is used to start workflows on each available processor." As a package that users directly interact with, and as a dependency of other packages in scientific Python, `multiprocessing` is a workhorse.

Finally we consider Dask, a Python package for task-based parallelism and analytics at scale. Users are increasingly interested in cluster runtimes where they queue up work, submit the work to the scheduler as a task graph, and the scheduler handles dependencies and farms out the tasks to workers. Dask also inter-operates with GPU analytics libraries from NVIDIA as part of RAPIDS, so we are naturally interested in its potential for our next system based in part on GPUs. As noted, large jobs using Dask are generally smaller than those using `mpi4py` (500 nodes versus 3000+ nodes), which may indicate a potential gap in scalability on Cori. The correlation data shown in Fig. 7 (right) indicate an affinity with the weather and climate community, where `netCDF4` and `xarray` seem particularly important. We reached out to several Dask users to learn more. One responded: "I don't remember having any Python Dask-related jobs running in the past 3 months." After some additional discussion and analysis, we discovered the user was using `xarray` which we believe was using Dask unbeknownst to the user. This kind of response from "Dask users" was not uncommon.

Discussion

Our results demonstrate that we are able to collect useful data on Python package use on Cori, tag it with additional metadata useful for filtering during analysis, and conduct exploratory analysis of the data that can easily evolve to production and publication. The results themselves confirm many of our expectations about Python use on Cori, but also reveal some surprises that suggest next actions for various stakeholders. Such surprises suggest new opportunities for engagements between NERSC, users, vendors, and developers of scientific Python infrastructure.

We observe that Python jobs on Cori mostly come from environments that users themselves have provisioned, and not directly from the Python software environment module that NERSC provides. Our expectation was that the fraction of jobs running from such environments would be high since we knew through interacting with our users that custom Conda environments were very popular. A major driver behind this popularity is that users often want versions of packages that are newer than they what can get from a centrally-managed Python environment. But rather than take that as a cue that we should be updating the NERSC-provided Python environment more often, finding new ways to empower users to manage their own software better has become our priority instead. This currently includes continuing to provide easy access to Conda environments, locations for centralized installations (i.e. shared by a collaboration), and improved support for containerized environments. However we are constantly reevaluating how best to support the needs of our users.

Other results indicate that this may need to be done carefully. As mentioned in the Results, only about 17% of jobs that use NumPy, SciPy, scikit-learn, or NumExpr are using versions of those packages that rely on OpenMP-threaded, optimized Intel MKL. Given that Cori's CPU architectures come from Intel, we might expect the best performance to come from libraries optimized for that architecture. We caution that there are a number of hypotheses to consider behind this observation, as it is a question of how well-educated users are on the potential benefits of such libraries. The surprising reliance of our users on `multiprocessing` and the tendency of users to use `mpi4py` for embarrassing parallelism suggest that users may find it easier to manage process parallelism than OpenMP thread parallelism in scientific Python. Another consideration is that users value ease in software installation rather than performance. Many Conda users rely heavily on the `conda-forge` channel, which does have a much greater diversity of packages as compared to the `defaults` channel, and will install libraries based on OpenBLAS. Users may be willing or able to tolerate some performance loss in favor of being able to easily install and update their software stack. (There are no easy answers or quick fixes to this problem of facilitating both easy installation and good performance, but this is a major goal of our efforts to support Python at NERSC.) Finally, many users install complex packages designed for use on a wide range of systems; many of these packages such as GPAW may use OpenBLAS rather than MKL. Having seen that MKL adoption is low, our goal is to try to better understand the factors leading to this and ensure that users who can benefit from MKL make good choices about how they build their Python environments through documentation, training, and direct recommendation.

While some discoveries suggest next actions and user engagement for NERSC staff, others suggest opportunities for broader stakeholder action. The importance of `multiprocessing` to users on nodes with large core count suggests an opportunity for developers and system vendors. Returning to the observation that jobs using AstroPy have an tendency to also use `mpi4py`, we conclude that users of AstroPy have been able to scale their AstroPy-based applications using MPI and that AstroPy developers may want to consider engaging with our users to make that interaction better. Examining the jobs further we find that these users tend to be members of large cosmology experiments like Dark Energy Survey [Abb18], Dark Energy Spectroscopic Instrument [DESI], the Dark Energy Science Collaboration [DESC], and CMB-S4 [Aba16]. The pattern appears over many users in several experiments. We also note that the use of `astropy.io.fits` in MPI-enabled Python jobs by astronomers suggests that issues related to FITS I/O performance in AstroPy on HPC systems may be another area of focus.

While the results are interesting, making support decisions based on data alone has its pitfalls. There are limitations to the data set, its analysis, and statements we can make based on the data, some of which can be addressed easily and others not. First and foremost, we address the limitation that we are tracking a prescribed list of packages, an obvious source of potential bias. The reason for prescribing a list is technical: Large bursts of messages from jobs running on Cori at one time caused issues for OMNI infrastructure and we were asked to find ways to limit the rate of messages or prevent such kinds of bursts. Since then, OMNI has evolved and may be able to handle a higher data rate, making it possible to simply report all entries in `sys.modules` excluding built-in and standard modules (but not entirely, as

`multiprocessing` would go undetected). One strategy may be to forward `sys.modules` to OMNI on a very small random subset of jobs (say 1%) and use that control data set to estimate bias in the tracked list. It also helps us control for a major concern, that of missing out on data on emerging new packages.

Another source of bias is user opt-out. Sets of users who opt out tend to do so in groups, in particular collaborations or experiments who manage their own software stacks: Opting out is not a random error source, it is another source of systematic error. A common practice is for such collaborations to provide scripts that help a user "activate" their environment and may unset or rewrite `PYTHONPATH`. This can cause undercounts in key packages, but we have very little enthusiasm for removing the opt-out capability. Rather, we believe we should make a positive case for users to remain opted in, based on the benefits it delivers to them. Indeed, that is a major motivation for this paper.

A different systematic undercount may occur for applications that habitually run into their allocated batch job wallclock limit. As mentioned with TensorFlow, we confirmed with users a particular pattern of submitting chains of dozens of training jobs that each pick up where the previous job left off. If all these jobs hit the wallclock limit, we will not collect any data. Counting the importance of a package by the number of jobs that use it is dubious; we favor understanding the impact of a package from the breadth of the user community that uses it. This further supports the idea that multiple approaches to understanding Python package use are needed to build a complete picture; each has its own shortcomings that may be complemented by others.

Part of the power of scientific Python is that it enables its developers to build upon the work of others, so when a user imports a package it may import several other dependencies. All of these libraries "matter" in some sense, but we find that often users are importing those packages without even being aware they are being used. For instance, when we contacted users who appeared to be running Dask jobs at a node count of 100 or greater, we received several responses like "I'm a bit curious as to why I got this email. I'm not aware to have used Dask in the past, but perhaps I did it without realizing it." More generally, large-scale jobs may use Python only incidentally for housekeeping operations. Importing a package is not the same as actual use, and use of a package in a job running at scale is not the same as that package actually being used at scale.

Turning to what we learned from the process of building our data analysis pipeline, we found that the framework gave us ways to follow up on initial clues and then further productionize the resulting exploratory analysis. Putting all the steps in the analysis (extraction, aggregation, indexing, selecting, plotting) into one narrative improves communication, reasoning, iteration, and reproducibility. One of our objectives was to manage as much of the data analysis as we could using one notebook for exploratory analysis with Jupyter, parameterized calculations in production with Papermill, and shared visualization as a Voilà dashboard. Using cell metadata helped us to manage both the computationally-intensive "upstream" part of the notebook and the less expensive "downstream" dashboard within a single file. One disadvantage of this approach is that it is very easy to remove or forget to apply cell tags. This could be addressed by making cell metadata easier to apply and manage. The Voilà JupyterLab extension helps with this problem by providing a preview of a dashboard rendering before it is published to the web. Another issue with the single-notebook pattern is that some code may be repeated for different purposes. This is not a source of error necessarily, but it can cause confusion. All of these issues disappear if the same hardware could be used to run the notebook in exploratory analysis, pipelined production, and dashboard phases, but these functions are simply not available in a single system at NERSC today.

Conclusion

We have taken our first small steps in understanding the Python workload at NERSC in detail. Instrumenting Python to record how frequently key scientific Python packages are being imported in batch jobs on Cori confirmed many of our assumptions but yielded a few surprises. The next step is acting on the information we have gathered, and of course, monitoring the impact those actions have.

Using Python itself as a platform for analyzing the Python workload poses a few challenges mostly related to supporting infrastructure and tooling. With a few tricks, we find that the same Jupyter notebooks can be used for both exploratory and production data analysis, and also to communicate high-level results through dashboards. We initiated this project not only to perform Python workload analysis but to test the supposition that users could assemble all the pieces they needed for a Python-based data science pipeline at NERSC. Along the way, we identified shortcomings in our ecosystem, and this motivated us to develop tools for users that fill those gaps, and gave us direct experience with the very same tools our users use to do real science.

In the near future, we will expand Python workload analysis to Perlmutter, a new system with CPU+GPU and CPU-only nodes to identify users of the CPU nodes who might be able to take advantage of GPUs. Other future plans include examining Python use within the context of specific science areas by linking our data with user account and allocation data, and using natural language processing and machine learning to proactively identify issues that users have with Python on our systems. Another interesting avenue to pursue is whether the monitoring data we gather may be of use to users as an aid for reproducible computational science. If users are able to access Python usage data we collect from their jobs, they could use it to verify what Python packages and package versions were used and obtain some degree of software provenance for reproducing and verifying their results.

We anticipate that developers of scientific Python software may find the information we gather to be informative. Readers can view the public MODS Python dashboard at https://mods.nersc.gov/public/

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We thank our colleagues Brian Austin, Tiffany Connors, Aditya Kavalur, and Colin MacLean for discussions on workload analysis, process monitoring, and Python. We also thank the Vaex developers for their help and advice, and the Dask-cuDF and cuDF developers for their responsiveness to issues and advice on effective use of Dask-cuDF and cuDF. Finally we thank our users who were kind enough to provide feedback to us and allow us to use their quotes about how they are using Python at NERSC.

REFERENCES

[Aba16] K. N. Abazajian, et al., *CMB-S4 Science Book, First Edition*, 2016. <https://arxiv.org/abs/1610.02743>

[Abb18] T. M. C. Abbott, et al., *Dark Energy Survey year 1 results: Cosmological constraints from galaxy clustering and weak lensing* Physical Review D, 98, 043526, 2018. <https://doi.org/10.1103/PhysRevD.98.043526>

[Age14] A. Agelastos, et al., *Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications,* Proc. IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis, SC14, New Orleans, LA, 2014. <https://doi.org/10.1109/SC.2014.18>

[Agr14] K. Agrawal, et al., *User Environment Tracking and Problem Detection with XALT,* Proceedings of the First International Workshop on HPC User Support Tools, Piscataway, NJ, 2014. <http://doi.org/10.1109/HUST.2014.6>

[Bab19] Y. Babuji, et al., *Parsl: Pervasive Parallel Programming in Python,* 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Phoenix, AZ, 2019. <https://doi.org/10.1145/3307681.3325400>

[Bau19] E. Bautista, et al., *Collecting, Monitoring, and Analyzing Facility and Systems Data at the National Energy Research Scientific Computing Center,* 48th International Conference on Parallel Processing: Workshops (ICPP 2019), Kyoto, Japan, 2019. <https://doi.org/10.1145/3339186.3339213>

[DESI] The DESI Collaboration, *The DESI Experiment Part I: Science, Targeting, and Survey Design,* Science Final Design Report, <https://arxiv.org/abs/1611.00036>

[Eva15] T. Evans, A. Gomez-Iglesias, and C. Proctor, *PyTACC: HPC Python at the Texas Advanced Computing Center,* Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing, SC15, Austin, TX, 2015 <https://doi.org/10.1145/2835857.2835861>

[Fah10] M. Fahey, N Jones, and B. Hadri, *The Automatic Library Tracking Database,* Proceedings of the Cray User Group, Edinburgh, United Kingdom, 2010. <https://doi.org/10.1145/1838574.1838582>

[Fur91] J. L. Furlani, *Modules: Providing a Flexible User Environment,* Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V), San Diego, CA, 1991.

[Gam15] T. Gamblin, et al., *The Spack Package Manager: Bringing Order to HPC Software Chaos,* in Supercomputing 2015, SC15, Austin, TX, 2015. <https://doi.org/10.1145/2807591.2807623>

[Jac16] D. M. Jacobsen and R. S. Canon, *Shifter: Containers for HPC,* in Cray Users Group Conference (CUG16), London, United Kingdom, 2016

[Jai15] Jain, A., et al., *FireWorks: a dynamic workflow system designed for high-throughput applications.* Concurrency Computat.: Pract. Exper., 27: 5037–5059, 2015. <https://doi.org/10.1002/cpe.3505>

[DESC] LSST Dark Energy Science Collaboration, *Large Synoptic Survey Telescope: Dark Energy Science Collaboration,* White Paper, 2012. <https://arxiv.org/abs/1211.0310>

[Mac17] C. MacLean. *Python Usage Metrics on Blue Waters* Proceedings of the Cray User Group, Redmond, WA, 2017.

[vaex] A. Maarten. and J. V. Breddels, *Vaex: big data exploration in the era of Gaia,* Astronomy & Astrophysics, 618, A13, 2018. <https://arxiv.org/abs/1801.02638v1>

[Mcl11] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, *Best practices for the deployment and management of production HPC clusters* in State of the Practice Reports, SC11, Seattle, WA, 2011. <https://doi.acm.org/10.1145/2063348.2063360>

[MODS] NERSC 2017 Annual Report. pg 31. <https://www.nersc.gov/assets/Uploads/2017NERSC-AnnualReport.pdf>

[OA20] NERSC Operational Assessment. In press, 2020.

Training machine learning models faster with Dask

Joesph Holt[‡], Scott Sievert[‡*]

Abstract—Machine learning (ML) relies on stochastic algorithms, all of which rely on gradient approximations with "batch size" examples. Growing the batch size as the optimization proceeds is a simple and usable method to reduce the training time, provided that the number of workers grows with the batch size. In this work, we provide a package that trains PyTorch models on Dask clusters, and can grow the batch size if desired. Our simulations indicate that for a particular model that uses GPUs for a popular image classification task, the training time can be reduced from about 120 minutes with standard SGD to 45 minutes with a variable batch size method.

Index Terms—machine learning, model training, distributed computation

Introduction

Training deep machine learning models takes a long time. For example, training a popular image classification model [RRSS19] to reasonable accuracy takes "around 17 hours" on Google servers.[1] Another example includes training an NLP model for 10 days on 8 high-end GPUs [RNSS18].[2] Notably, the number of floating point operations (FLOPs) required for "the largest AI training runs" doubles every 3.4 months.[3]

Model training is fundamentally an optimization problem: it tries to find a model $\widehat{\boldsymbol{w}}$ that minimizes a loss function F:

$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} F(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{w}; \boldsymbol{z}_i)$$

where there are n examples in the training set, and each example is represented by $\boldsymbol{z}_i$. For classification, $\boldsymbol{z}_i = (\boldsymbol{x}_i, y_i)$ for a label y_i and feature vector $\boldsymbol{x}_i$. The loss function F is the mean of the loss f over different examples. To compute this minimization for large scale machine learning, stochastic gradient descent (SGD) or a variant thereof is used [BCN18]. SGD is iterative, and the model update at each step k is computed via

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \frac{\gamma_k}{B_k}\sum_{i=1}^{B_k} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_{i_s})$$

where $\boldsymbol{g}$ is the gradient of the loss function f for some batch size $B_k \geq 1$, i_s is chosen uniformly at random and $\gamma_k > 0$ is the learning rate or step size. The objective function's gradient is approximated with B_k examples – the gradient approximation $\frac{1}{B_k}\sum_{i=1}^{B_k} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_{i_s})$ is an unbiased estimator of the loss function F's gradient. This computation is common in the vast majority of SGD variants, and is found in popular variants like Adam [KB14], RMSprop [ZSJ+19], Adagrad [DHS11], Adadelta [Zei12], and averaged SGD [PJ92]. Most variants make modifications to the learning rate γ_k [DHS11], [Zei12], [KB14], [ZSJ+19].

Increasing the batch size B_k will reduce the number of model updates while not requiring more FLOPs or gradient computations – both empirically [SKYL17] and theoretically [Sie20]. Typically, the number of FLOPs controls the training time because training is performed with a single processor. At first, fewer model updates seems like an internal benefit that doesn't affect training time.

The benefit comes when training with multiple machines, aka a distributed system. Notably, the time required to complete a single model update is (nearly) agnostic to the batch size provided the number of workers in a distributed system grows with the batch size. In one experiment, the time to complete a model update grows by 13% despite the batch size growing by a factor of 44 [GDG+17, Sec. 5.5]. This acceleration has also been observed with an increasing batch size schedule [SKYL17, Sec. 5.4].

Contributions

We provide software to accelerate machine learning model training, at least with certain distributed systems. For acceleration, the distributed system must be capable of assigning a different number of workers according to a fixed schedule. Specifically, this work provides the following:

- A Python software package to train machine learning models. The implementation[4] provides a Scikit-learn API [BLB+13] to PyTorch models [PGM+19].
- Our software works on any cluster that is configured to work with Dask, many of which can change the number of workers on demand.[5]
- Extensive experiments to illustrate that our software can accelerate model training in terms of wall-clock time when an appropriate Dask cluster is used.

A key component of our software is that the number of workers grows with the batch size. Then, the model update time is agnostic to the batch size provided that communication is instantaneous. This has been shown empirically: Goyal et al. grow the batch

‡ *University of Wisconsin–Madison*
* *Corresponding author: stsievert@wisc.com*

1. Specifically, a ResNet-50 model on the ImageNet database using a Google Tensor Proceesing Unit (TPU) (github.com/tensorflow/tpu/.../resnet/README.md).

2. See OpenAI's blog post "Improving Language Understanding with Unsupervised Learning."

3. See OpenAI's blog post "AI and Compute."

4. https://github.com/stsievert/adadamp

5. Including the default usage (through LocalCluster), supercomputers (through Dask Job-Queue), YARN/Hadoop clusters (through Dask Yarn) and Kubernetes clusters (through Dask Kubernetes).

size (and the number of workers with it) by a factor of 44 but the time for a single model update only increases by a factor of 1.13 [GDG+17, Sec. 5.5].

Now, let's cover related work to gain understanding of why variable batch sizes provide a benefit in a distributed system. Then, let's cover the details of our software before presenting simulations. These simulations confirm that model training can be accelerated if the number of workers grows with the batch size. Methods to workaround limitations on the number of workers will be presented.

Related work

The data flow for distributed model training involves distributing the computation of the gradient estimate, $\frac{1}{B}\sum_{i=1}^{B} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_i)$. Typically, each worker computes the gradients for B/P examples when there is a batch size of B and P machines. Then, the average of these gradients is taken and the model is updated.[6]

Clearly, Amdahl's law is relevant because there are diminishing returns as the number of workers P is increased [GVY+18]. This as referred to as "strong scaling" because the batch size is fixed and the number of workers is treated as an internal detail. By contrast, growing the amount of data with the number of workers is known as "weak scaling." Of course, relevant experiments show that weak scaling exhibits better scaling than strong scaling [QST17].

Constant batch sizes

To circumvent Amdahl's law, a common technique is to increase the batch size [ZLN+19] alongside the learning rate [JAGG20]. Using moderately large batch sizes yields high quality results more quickly and, in practice, requires no more computation than small batch sizes, both empirically [GDG+17] and theoretically [YPL+18].

There are many methods to choose the best constant batch size (e.g., [GGS19], [KSL+20]). Some methods are data dependent [YPL+18], and others depend on the model complexity. In particular, one method uses hardware topology (e.g., network bandwidth) in a distributed system [PKK+19].

Large constant batch sizes present generalization challenges [GDG+17]. The generalization error is hypothesized to come from "sharp" minima, strongly influenced by the learning rate and noise in the gradient estimate [KMN+16]. To match performance on the training dataset, careful thought must be given to hyperparameter selection [GDG+17, Sec. 3 and 5.2]. In fact, this has motivated algorithms specifically designed for large constant batch sizes and distributed systems [JAGG20], [JSH+18], [YGG17].

Increasing the batch size

Model quality greatly influences the amount of information in the gradient – which influences the batch size [Sie20]. For example, if models are poorly initialized, then using a large batch size has no benefit: the gradient—or direction to the optimal model—for each example will produce very similar numbers. An illustration is given in Figure 1.

Various methods to *adaptively* change the batch size based on model performance have been proposed [Sie20], [DYJG16], [BRH17], [BCNW12]. Of course, these methods are adaptive so computing the batch size requires computation (though there are workarounds [Sie20], [BRH17]).

Convergence results have been given for adaptive batch sizes [Sie20], [BCN18], [ZYF18]. Increasing the batch size is a provably good measure that requires far fewer model updates and no more computation than standard SGD for strongly convex functions [BCN18, Ch. 5], and all function classes if the batch size is provided by an oracle [Sie20]. Convergence proofs have also been given for the *passively* increasing the batch size, both for strongly convex functions [BCN18, Ch. 5] and for non-convex functions [ZYF18]. Both of these methods require fewer model updates than SGD *and* do not increase the number of gradient computations.

Notably, a geometric batch size increase schedule has shown great empirical performance in image classification [SKYL17]. Specifically, the number of model updates required to finish training decreased by a factor of 2.2 over standard SGD [SKYL17]. Smith et al. make an observation that increasing the batch size and decreasing the learning rate both decay the optimization's "noise scale" (or variance of the model update), which has connections to simulated annealing [SKYL17]. This motivates increasing the batch size by the same factor the learning rate decays [SKYL17].

Both growing the batch size and using large constant batch sizes should require the same number of floating point operations as using standard SGD with small batch sizes to reach a particular training loss (respectively [Sie20], [BCN18] and [JAGG20], [YLR+19], [YPL+18]). Some proof techniques suggest that variable batch size methods mirror gradient descent [Sie20], [KNS16], so correspondingly, the implementations do not require much additional hyperparameter tuning [SKYL17].

Distributed training with Dask

We have written "AdaDamp," a software package to to train a PyTorch model with a Scikit-learn API on any Dask cluster.[7] It supports the use of constant or variable batch sizes, which fits nicely with Dask's ability to change the number of workers.[8] In this section, we will walk through the basic architecture of our software and an example usage. We will defer showing the primary benefit of our software to the experimental results.

Architecture

Our software uses a centralized synchronous parameter server and controls the data flow of the optimization with Dask (and does not rely on PyTorch's distributed support). Specifically, the following happen on every model update:

1) The master node broadcasts the model to every worker.
2) The workers calculate the gradients.
3) The workers communicate the gradients back to the master.
4) The master performs a model update with the aggregated gradients.

We use Dask to implement this data flow, which adds some overhead.[9] AdaDamp supports static batch sizes; however, there is little incentive to use AdaDamp with a static batch sizes: the

6. Related but tangential methods include methods to efficiently communicate the gradient estimates [AGL+17], [GTAZ18], [WSL+18].

7. While our software works with a constant batch size, the native implementations work with constant batch sizes and very likely have less overhead (e.g., PyTorch Distributed [LZV+20]).

8. https://github.com/stsievert/adadamp

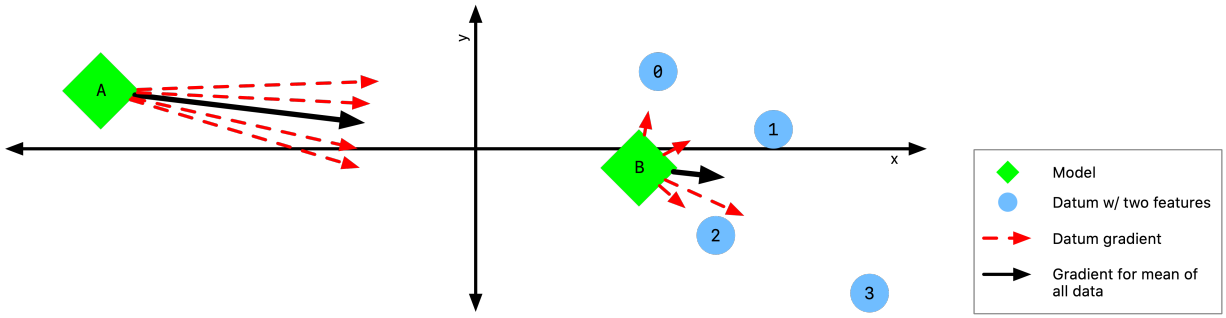

Fig. 1: *An illustration of why the batch size should increase. Here, let's find a model* $\mathbf{w} = [w_x, w_y]$ *that minimizes the function* $f(w_x, w_y) = \sum_{i=0}^{3}(w_x - x_i)^2 + (w_y - y_i)^2$ *where* x_i *and* y_i *are the x and y coordinates of each datum. When closer to the optimum at model A, the gradients are more "diverse," so the magnitude and orientation of each datum's gradient varies more [YPL+18].*

native solution has PyTorch less overhead [LZV+20], and already has a Dask wrapper.[10]

The key component of AdaDamp is that the number of workers grows with the batch size. Then, the model update time is agnostic to the batch size (provided communication is instantaneous). This has been shown empirically: Goyal et al. grow the batch size (and the number of workers with it) by a factor of 44 but the time for a single model update only increases by a factor of 1.13 [GDG+17, Sec. 5.5].

Example usage

First, let's create a standard PyTorch model. This is a simple definition; a more complicated model or one that uses GPUs can easily be substituted.

```
import torch.nn as nn
import torch.nn.functional as F

class HiddenLayer(nn.Module):
    def __init__(self, features=4, hidden=2, out=1):
        super().__init__()
        self.hidden = nn.Linear(features, hidden)
        self.out = nn.Linear(hidden, out)

    def forward(self, x, *args, **kwargs):
        return self.out(F.relu(self.hidden(x)))
```

Now, let's create our optimizer:

```
from adadamp import DaskRegressor
import torch.optim as optim

est = DaskRegressor(
    module=HiddenLayer, module__features=10,
    optimizer=optim.Adadelta,
    optimizer__weight_decay=1e-7,
    max_epochs=10
)
```

So far, a PyTorch model and optimizer have been specified. As per the Scikit-learn API, we specify parameters for the model/optimizer with double underscores, so in our example `HiddenLayer(features=10)` will be created. We can set the batch size increase parameters at initialization if desired, or inside `set_params`.

```
from adadamp.dampers import GeoDamp
est.set_params(
    batch_size=GeoDamp, batch_size__delay=60,
    batch_size__factor=5)
```

This will increase the batch size by a factor of 5 every 60 epochs, which is used in the experiments. Now, we can train:

```
from sklearn.datasets import make_regression
X, y = make_regression(n_features=10)
X = torch.from_numpy(X.astype("float32"))
y = torch.from_numpy(y.astype("float32")).reshape(-1, 1)

est.fit(X, y)
```

Experiments

In this section, we present two sets of experiments.[11] Both experiments will use the same setup, a Wide-ResNet model in a "16-4" architecture [ZK16] to perform image classification on the CIFAR10 dataset [KH09]. This is a deep learning model with about 2.75 million weights that requires a GPU to train.[12] The experiments will provide evidence for the following points:

1) Increasing the batch size reduces the number of model updates.
2) The time required for model training is roughly proportional to the number of model updates (presuming the distributed system is configured correctly).

To provide evidence for these points, let's run one set of experiments that varies the batch size increase schedule. These experiments will mirror the experiments by Smith et al. [SKYL17]. Additionally, let's ensure that our software accelerates model training as the number of GPUs increase.

We train each batch size increase schedule once, and then write the historical performance to disk. This reduces the need for many GPUs, and allows us to simulate different networks and highlight the performance of Dask. That means that in our simulations, we simulate model training by having the computer sleep for an appropriate and realistic amount of time.

9. An opportunity for future work.
10. https://github.com/saturncloud/dask-pytorch-ddp
11. Full detail on these experiments can be found at https://github.com/stsievert/adadamp-experiments
12. Specifically, we used a NVIDIA T4 GPU with an Amazon `g4dn.xlarge` instance. Training consumes 2.2GB of GPU memory with a batch size of 32, and 5.5GB with a batch size of 256.

Batch size increase

To illustrate the primary benefit of our software, let's perform several trainings that require a different number of model updates. These experiments explicitly mirror the experiments by Smith et al. [SKYL17, Sec. 5.1], which helps reduce the parameter tuning.

Largely, the same hyperparameters are used. These experiments only differ in the choice of batch size and learning rate, as shown in Figure 2. As in the Smith et al. experiments, every optimizer uses Nesterov momentum [Nes98] and the same momentum (0.9) and weight decay ($0.5 \cdot 10^{-3}$). They start with the same initial learning rate (0.05),[13] and either the learning rate is decreased or the batch size increases by a specified factor (5) at particular intervals (epochs 60, 120 and 180). This means that the variance of the model update is reduced by a constant factor at each update.

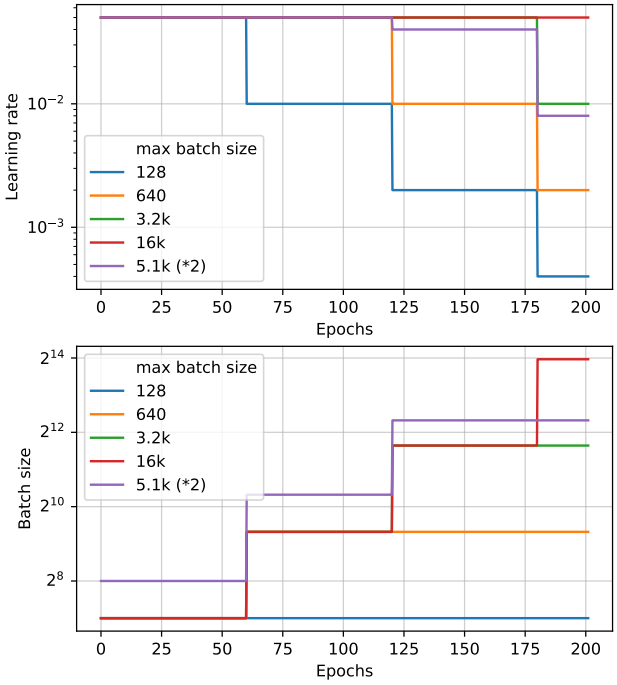

Fig. 2: *The learning rate and batch size decrease/increase schedules for various optimizers. After the maximum batch size is reached, the learning rate decays. A postfix of "(*2)" means the initial batch size twice as large (256 instead of 128)*

These different decay schedules exhibit the same performance in terms of number of epochs, which is proportional to the number of FLOPs, as shown in Figure 3. The number of FLOPs is (approximately) to the cost, at least on Amazon EC2 where the cost to rent a server tends to be proportional to the number of GPUs.

Importantly, this work focuses on increasing the number of workers with the batch size – the effect of which is hidden in Figure 3. However, the fact that the performance does not change with different schedules means that choosing a different batch size increase schedule will not require more wall-clock time if only a single worker is available. Combined with the hyperparameter similarity between the different schedules, this reduces deployment and debugging concerns.

13. These are the same as Smith et al. [SKYL17] with the exception of learning rate (which had to be reduced by a factor of 2).

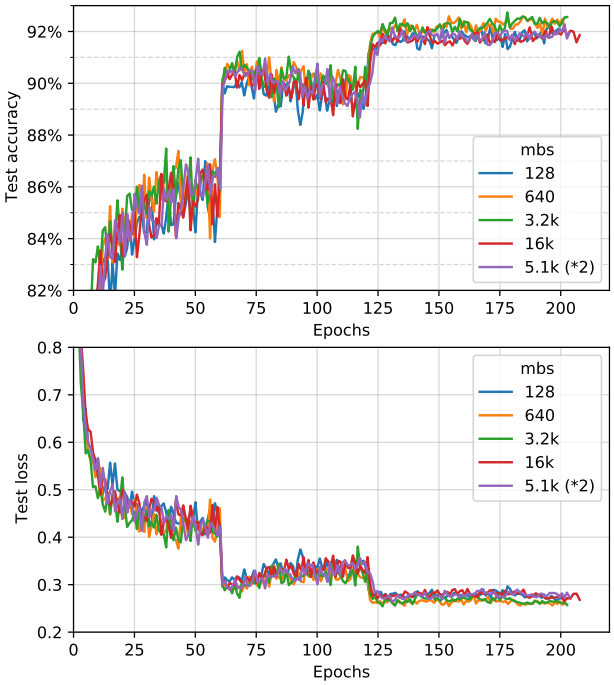

Fig. 3: *The performance of the LR/BR schedules in Figure 2, plotted with epochs—or passes through the dataset—on the x-axis.*

Maximum batch size	Model updates	Training time (min)	Max. workers
5.1k (*2)	14,960	69.87	40
3.2k	29,480	107.17	25
16k	29,240	107.49	125
640	34,520	116.86	5
128	78,200	200.19	1

TABLE 1: *A summary of the simulations in Figures 3 and 4. All training require approximately 200 epochs, so they all require the same number of FLOPs.*

If the number of workers grows with the batch size, then the number of model updates is relevant to the wall-clock time. Figure 4 shows the number of model updates and wall-clock time required to reach a model of a particular test accuracy. Of course, there is some overhead to our current framework, which is why the number of model updates does not exactly correlate with the wall-clock time required to complete training. In summary, the time required to complete training is shown in Table 1.

Future work

Architecture

Fundamentally, the model weights can be either be held on a master node (centralized), or on every node (decentralized). Respectively, these storage architectures typically use point-to-point communication or an "all-reduce" communication. Both centralized [LAP+14], [ABC+16] and decentralized [LZV+20], [SDB18] communication architectures are common.

Future work is to avoid the overhead introduced by manually having Dask control the model update workflow. With any synchronous centralized system, the time required for any one model update is composed of the time required for the following tasks:

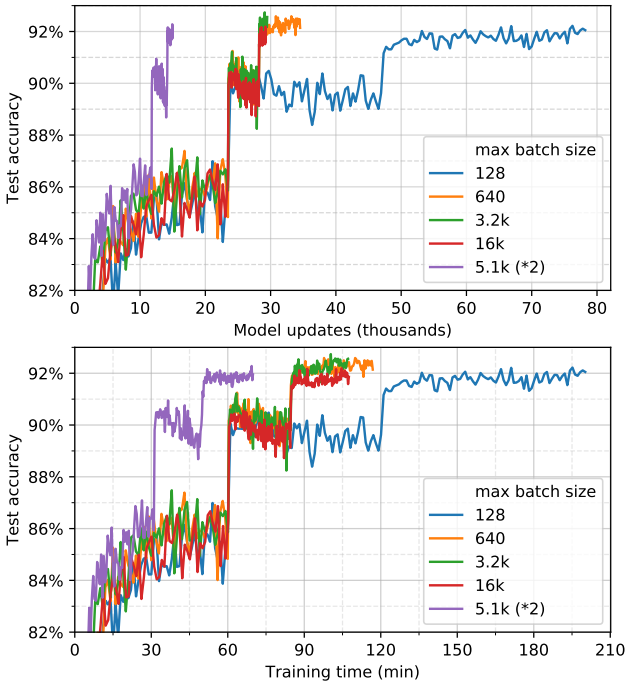

Fig. 4: *The same simulations as in Figure 3, but plotted with the number of model updates and wall-clock time plotted on the x-axis (the loss obeys a similar behavior as illustrated in the Appendix).*

1) Broadcasting the model from the master node to all workers
2) Finishing gradient computation on all workers.
3) Communicating gradients back to master node.
4) Various overhead tasks (e.g., serialization, worker scheduling, etc).
5) Computing the model update after all gradients are computed & gathered.

Items (1), (3) and (4) are a large concern in our implementation. Decentralized communication has the advantage of eliminating items (1) and (4), and mitigates (3) with a smarter communication strategy (all-reduce vs. point-to-point). Item (2) is still a concern with straggler nodes [DCM+12], but recent work has achieved "near-linear scalability with 256 GPUs" in a homogeneous computing environment [LZV+20]. Items (2) and (5) can be avoided with asynchronous methods (e.g., [RRWN11], [ZHA16]).

That is, most of the concerns in our implementation will be resolved with a distributed communication strategy. The PyTorch distributed communication package uses a synchronous decentralized strategy, so the model is communicated to each worker and gradients are sent between workers with an all-reduce scheme [LZV+20]. It has some machine learning specific features to reduce the communication time, including performing both computation and communication concurrently as layer gradients become available [LZV+20, Sec. 3.2.3].

The software library dask-pytorch-ddp[14] allows use of the PyTorch decentralized communications [LZV+20] with Dask clusters, and is a thin wrapper around PyTorch's distributed communication package. Future work will likely involve ensuring training can efficiently use a variable number of workers.

14. https://github.com/saturncloud/dask-pytorch-ddp

Maximum batch size	Centralized	Decentralized (moderate)	Decentralized (high)
5.1k (*2)	69.9	45.1	43.5
3.2k	107.2	67.7	65.5
16k	107.5	67.7	65.7
640	116.9	73.6	71.8
128	200.2	121.7	121.5

TABLE 2: *Simulations that indicate how the training time (in minutes) will change under different architectures and networks. The "centralized" architecture is the currently implemented architecture, and has the same numbers as "training time" in Table 4.*

Simulations

We have simulated the expected gain from the work of enabling decentralized communication with two networks that use a decentralized all-reduce strategy:

- `decentralized-medium` It assumes an a network with inter-worker bandwidth of 54Gb/s and a latency of 0.05μs.
- `centralized` uses a centralized communication strategy (as implemented) and the same network as `decentralized-medium`.
- `decentralized-high` has the same network as `decentralized-medium` but has an inter-worker bandwidth of 800Gb/s and a latency of 0.025μs.

To provide baseline performance, we also show the results with the current implementation:

- `centralized` uses the same network as `decentralized-medium` but with the centralized communication scheme that is currently implemented.

`decentralized-medium` is most applicable for clusters that have decent bandwidth between nodes. It's also applicable to for certain cases when Amazon EC2 is used with one GPU per worker,[15] or workers have a very moderate Infiniband setup.[16] `decentralized-high` is a simulation of the network used by the PyTorch developers to illustrate their distributed communication [LZV+20]. We have run simulations to illustrate the effects of these networks. Of course, changing the underlying networks does not affect the number of epochs or model updates, so Figures 3 and 4 also apply here.

A summary of how different networks affect training time is shown in Table 2. We show the training time for a particular network (`decentralized-moderate`) in Figure 6; `decentralized-high` shows similar performance as illustrated in Table 2. A visualization of 2 is shown in Figure 5. This shows how network quality affects the performance of different optimization methods in Figure 6. Clearly, the optimization method (and the maximum number of workers) is more important than the network.

Finally, let's show how the number of Dask workers affects the time required to complete a single epoch with a constant

15. 50Gb/s and 25Gb/s networks can be obtained with `g4dn.8xlarge` and `g4dn.xlarge` instances respectively. `g4dn.xlarge` machines have 1 GPU each and are the least expensive for a fixed number of FLOPs on the GPU.

16. A 2011 Infiniband setup with 4 links (https://en.wikipedia.org/wiki/InfiniBand#Performance)

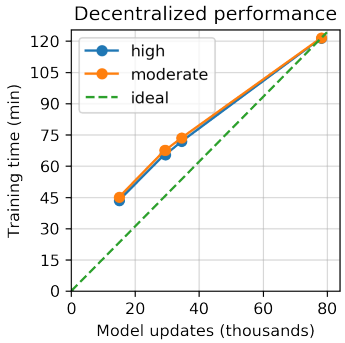

Fig. 5: *A single point represents one run in Figure 6. The point with about 80k model updates represents a single worker, so there's no overhead in this decentralized simulation. Different network qualities are shown with different colors, and the "ideal" line is as if every model update is agnostic to batch size.*

batch size. This simulation will use the `decentralized-high` network and has the advantage of removing any overhead. The results in Figure 7 show that the speedups start saturating around 128 examples/worker for the model used with a batch size of 512. Larger batch sizes will likely mirror this performance – computation is bottleneck with this model/dataset/hardware.

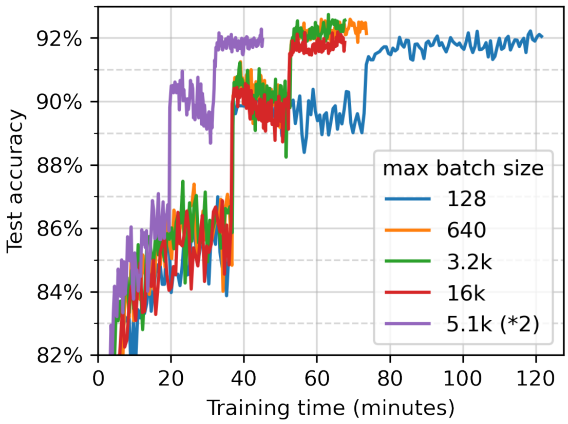

Fig. 6: *The training time required for different optimizers under the* `decentralized-moderate` *network.*

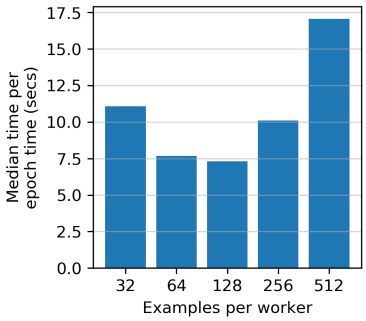

Fig. 7: *The median time to complete a pass through the training set with a batch size of 512. As expected, the speedups diminish when there is little computation and much communication (say with 32 examples per worker).*

Conclusion

In this work, we have provided a package to train PyTorch ML models with Dask cluster. This package reduces the amount of time required to train a model with the current centralized setup. However, it can be further accelerated by integration with PyTorch's distributed communication package as illustrated by extensive simulations. For a particular model, only 45 minutes is required for training – an improvement over the 120 minutes required with standard SGD.

APPENDIX

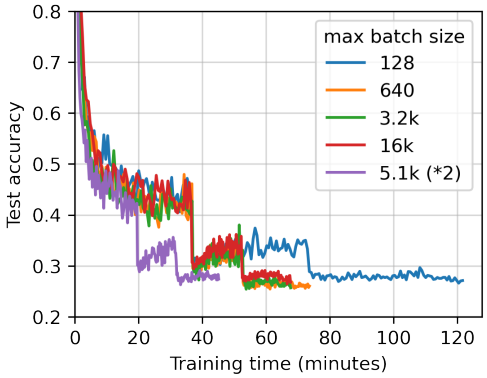

Fig. 8: *The training time required for different optimizers under the* `decentralized-moderate` *network.*

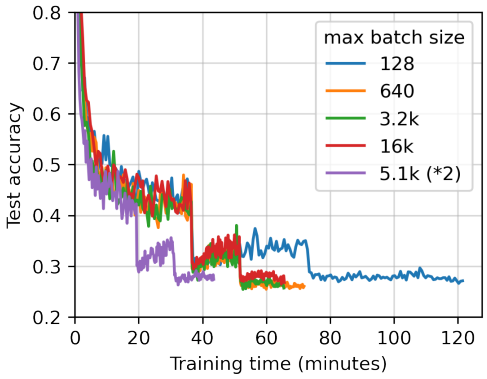

Fig. 9: *The training time required for different optimizers under the* `decentralized-high` *network.*

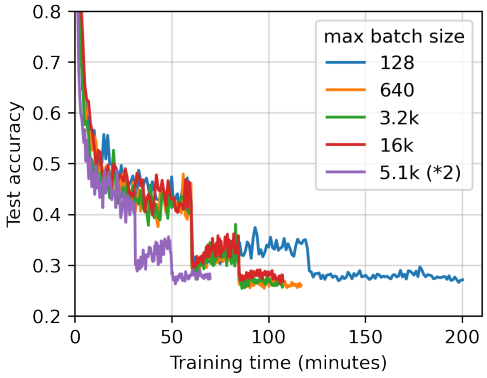

Fig. 10: *The training time required for different optimizers under the* `centralized` *network.*

REFERENCES

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/6c340f25839e6acdc73414517203f5f0-Paper.pdf.

[BCN18] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60:223–223, 2018. doi:10.1137/16M1080173.

[BCNW12] Richard H Byrd, Gillian M Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012. doi:10.1007/s10107-012-0572-5.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[BRH17] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. In *33rd Conference on Uncertainty in Artificial Intelligence (UAI 2017)*, pages 675–684. Curran Associates, Inc., 2017. URL: http://auai.org/uai2017/proceedings/papers/141.pdf.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. 25, 2012. URL: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL: http://jmlr.org/papers/v12/duchi11a.html.

[DYJG16] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Big Batch SGD: Automated inference using adaptive batch sizes. *arXiv preprint arXiv:1610.05792*, 2016.

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[GGS19] Nidham Gazagnadou, Robert Gower, and Joseph Salmon. Optimal mini-batch and step sizes for SAGA. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2142–2150. PMLR, 09–15 Jun 2019. URL: http://proceedings.mlr.press/v97/gazagnadou19a.html.

[GTAZ18] Demjan Grubic, Leo K Tam, Dan Alistarh, and Ce Zhang. Synchronous multi-gpu deep learning with low-precision communication: An experimental study. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 145–156. OpenProceedings, 2018. doi:10.3929/ethz-b-000319485.

[GVY+18] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

[JAGG20] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A user-friendly algorithm for distributed training. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4911–4920. PMLR, 13–18 Jul 2020. URL: http://proceedings.mlr.press/v119/johnson20a.html.

[JSH+18] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong

Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, and Liwei Yu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[KNS16] Hamed Karimi, Julie Nutini, and Mark Schmidt. Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, page 795–795, 2016. doi:10.1007/978-3-319-46128-1_50.

[KSL+20] Ahmed Khaled, Othmane Sebbouh, Nicolas Loizou, Robert M Gower, and Peter Richtárik. Unified analysis of stochastic gradient methods for composite convex and smooth optimization. *arXiv preprint arXiv:2006.11573*, 2020.

[LAP+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[LZV+20] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020. URL: https://doi.org/10.14778/3415478.3415530, doi:10.14778/3415478.3415530.

[Nes98] Yurii Nesterov. *Introductory lectures on convex programming volume i: Basic course*, volume 3. 1998.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[PJ92] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992. doi:10.1137/0330046.

[PKK+19] Michael P Perrone, Haidar Khan, Changhoan Kim, Anastasios Kyrillidis, Jerry Quinn, and Valentina Salapura. Optimal mini-batch size selection for fast gradient descent. *arXiv preprint arXiv:1911.06459*, 2019.

[QST17] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR (Poster)*, 2017. URL: https://talwalkarlab.github.io/paleo/.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[RRSS19] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 09–15 Jun 2019. URL: http://proceedings.mlr.press/v97/recht19a.html.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

[SDB18] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[Sie20] Scott Sievert. Improving the convergence of sgd through adaptive batch sizes, 2020.

[SKYL17] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc Vgra Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[WSL+18] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/33b3214d792caf311e1f00fd22b392c5-Paper.pdf.

[YGG17] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[YLR+19] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[YPL+18] Dong Yin, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. Gradient diversity: a key ingredient for scalable distributed learning. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1998–2007. PMLR, 09–11 Apr 2018. URL: http://proceedings.mlr.press/v84/yin18a.html.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[ZHA16] Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. pages 629–638, 2016. doi:10.1109/ICDM.2016.0074.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[ZLN+19] Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/e0eacd983971634327ae1819ea8b6214-Paper.pdf.

[ZSJ+19] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of adam and rmsprop. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11127–11135, 2019. doi:10.1109/cvpr.2019.01138.

[ZYF18] Pan Zhou, Xiaotong Yuan, and Jiashi Feng. New insight into hybrid stochastic gradient descent: Beyond with-replacement sampling and convexity. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/67e103b0761e60683e83c559be18d40c-Paper.pdf.

Multithreaded parallel Python through OpenMP support in Numba

Todd Anderson‡†, Tim Mattson‡*

Abstract—A modern CPU delivers performance through parallelism. A program that exploits the performance available from a CPU must run in parallel on multiple cores. This is usually best done through multithreading. Threads belong to a process and share the memory associated with that process. The most popular approach for writing multithreaded code is to use directives to tell the compiler how to convert code into multithreaded code. The most commonly used directive-based API for writing multithreaded code is OpenMP. Python is not designed for parallel programming with threads. The GlobalInterpreterLock (GIL) prevents multiple threads from simultaneously accessing Python objects. This effectively prevents data races and makes Python naturally thread safe. Consequently, the GIL prevents parallel programming with multiple threads and therefore keeps Python from accessing the full performance from a CPU. In this paper, we describe a solution to this problem. We implement OpenMP in Python so programmers can easily annotate their code and then let the Numba just-in-time (JIT) compiler generate multithreaded, OpenMP code in LLVM, thereby bypassing the GIL. We describe this new multithreading system for Python and and show that the performance in our early tests is on par with the analogous C code.

Index Terms—OpenMP, Python, Numba

Introduction

Python emphasizes productivity over performance. Given the growth of Python and the fact it is by some measures the most popular programming language in the world [PyP21] , this focus on productivity has turned out to be the right choice at the right time.

Performance from Python code, however, has suffered. The recent paper about software performance ("There's plenty of room at the top..." [Lei20]) used a simple triply nested loop for a matrix multiply routine and found that the Python code delivered results that, when rounded to the correct number of significant digits, were zero percent of the available peak performance from the chip.

A common attitude in the high performance computing community is that Python is for developing new algorithms or managing workflows built up from external, high performance modules written in low level languages. If the performance is not good enough from Python, then the code is reimplemented in a low-level language such as C. Why sacrifice productivity for performance if programmers who need high performance will rewrite their code in C anyway?

In our line of research, we are developing technologies that let programmers stay in Python. Some of our prior work in this area is ParallelAccelerator in Numba (i.e., the parallel=True option to the Numba JIT decorator) ([Lam15] [And17] [NPar]) In this work, common patterns in code are exploited to expose concurrency in the code which is then executed in parallel. The parallelism is implicit in the patterns of code written by a programmer aware of these tools. Implicit parallelism is powerful but there are too many cases where it can not find sufficient concurrency to support the levels of parallelism needed to fully occupy the cores on a modern CPU.

Another approach is to embed parallelism inside the functions from modules such as NumPy. This is an effective way to exploit the parallel resources of a system. However, there are two well known limitations to this approach. First, if the parallelism is constrained to the body of a function, there is startup overhead in launching the parallel threads and shutting them down for each function call. This overhead may be small, but they add up since they occur each time a function is called. This increases the fraction of a program execution time that is not reduced as parallel resources are added (the "serial fraction") and limits the maximum possible speedup (which is restricted by Amdahl's law to one over the serial fraction). Second, limiting parallelism to the bodies of parallelized functions misses the opportunity for additional parallelism that comes from running those functions in parallel. A great deal of available parallelism is missed if such cross-function parallelism is not exploited.

Taken together, we believe these limitations to parallelism strongly suggest that a well-rounded Python tool chain needs to include explicit parallelism. With our focus on programming the cores in a CPU, this translates into utilizing multiple threads in parallel from python code. The problem is that the GlobalInterpreterLock (GIL) prevents multiple threads from simultaneously accessing Python objects. The GIL helps programmers write correct code. It reduces the chances that a program would produce different results based on how threads are scheduled (a "race condition"). It means that loads and stores from memory are unlikely to conflict and create a "data race". There were very good reasons for including the GIL in Python. It has the effect, however, of preventing parallel programming with multiple threads and therefore keeps Python from accessing the full performance from a CPU.

Multithreading for performance has been a foundational technology for writing applications in high performance computing

† *These authors contributed equally.*
‡ *Intel Corp.*
* *Corresponding author: timothy.g.mattson@intel.com*

for over four decades. Its use grew rapidly in 1997 with the introduction of OpenMP [deS18] . This API worked with C, C++ and Fortran compilers so a programmer could direct the compiler to generate multithreaded code through a sequence of directives. The power of OpenMP is that a programmer can add parallelism to a serial program incrementally, evolving a program step by step into a parallel program.

In this paper, we introduce PyOMP: a research prototype system with support for OpenMP directives in Python. PyOMP uses the Python with statement to expose the directives in OpenMP. These with statements are interpreted by our custom Numba JIT and combined with a backend that connects these constructs to analogous entry points in the generated LLVM code. This LLVM code is then compiled using the Intel LLVM system, which includes support for the full range of OpenMP constructs. For this early research prototype, we restrict ourselves to a subset of OpenMP known as the OpenMP Common Core [Mat19] . We describe the subset of OpenMP supported from Python including the most common design patterns used by OpenMP programmers. We then discuss details of how we worked with Numba to implement this tool. Finally, we include some preliminary benchmark numbers and then close with a description of our future plans for PyOMP.

We want to be clear about what this paper is not. It is not a review of alternative approaches for parallel programming in Python. It is not a benchmarking paper where we compare major approaches for parallel programming in Python. We hope to write those papers in the future, but that is work-in-progress. Furthermore, we take benchmarking very seriously. We will not publish benchmarks that have not been fully optimized, reviewed for correctness by the authors of the alternative systems we are comparing against, and address patterns application developers actually use. A good benchmarking paper is a major undertaking, which we fully intend to do. For now, however, we just want to introduce this new form of parallel programming in Python; to gauge interest and find fellow-travelers to join us as we turn this research prototype system into a robust technology for the general Python programming community.

PyOMP: Python and OpenMP

OpenMP is a standard API that defines a set of compiler directives that can be used with C, C++ and Fortran to help applications programmers write multithreaded code. First released in 1997, it continues to be the most common way programmers in the high performance computing community write multithreaded code.

The standard has grown in size and complexity as it has evolved from version 1.0 to the current release (version 5.1). Important new features have been added over the years including support for programming GPUs and detailed controls over how a computation is mapped onto many cores in a nonuniform address space. Most OpenMP programmers, however, restrict themselves to a subset of the OpenMP 3.0 specification released in 2008. This subset of the 21 most commonly used elements of OpenMP is called the "OpenMP Common Core" [Mat19] .

PyOMP is a research prototype system implementing OpenMP in Python. In PyOMP, we cover 90% of the common core. These are are summarized in table I. PyOMP is tied to the Numba JIT (Just-In-Time) compilation system. Any function using PyOMP must be JIT'ed with Numba. The contents of PyOMP are provided as a module included with Numba. The Numba compiler works with NumPy arrays which must be used for any arrays inside a PyOMP function.

The essence of OpenMP is the well-known fork-join parallelism foundational to most multithreaded programming models. A program begins as a serial thread. At some point, a compute intensive block of work is encountered. If this can be broken down into a set of tasks that can run at the same time AND unordered with respect to each other (in other words, they are concurrent), a team of threads is forked to do this work in parallel. When the threads are done, they join together and the original serial thread continues.

In essence, an OpenMP program is a sequence of serial and parallel executions. The API is expressed in terms of directives to the compiler which handles the tedious work of packaging code into functions for the threads, managing threads, and synchronizing the threads to maintain a consistent view of memory. The programming style is one of incremental parallelism so a program evolves in phases from a serial program into a parallel program.

Obviously, a detailed course on OpenMP is well beyond the scope of this paper. Instead, we present the three core design patterns used in OpenMP. These are SPMD (Single Program Multiple Data), Loop Level Parallelism, and Divide and Conquer with tasks. We will describe each of these patterns in turn and in doing so describe the key elements of PyOMP. We will apply these patterns to a single problem; the numerical integration of *4/(1+x^2)* from zero to one. If the program is correct, the result of this definite integral should be an approximation of pi.

The SPMD Pattern

A classic use of the SPMD pattern is shown in figure 1. In an SPMD pattern, you create a team of threads and then, using the rank of a thread (a number ranging from zero to the number of threads minus one) and the number of threads, explicitly control how work is divided between the threads. Threads are created with the parallel construct expressed in PyOMP using the with context statement. We see this in line 14-15. The identifier openmp indicates this is an element of PyOMP and parallel indicates that the compiler should fork a team of threads. These threads come into "existence" at that point in the program and they each redundantly execute the work in the code associated with the with statement. This code is called a structured block in OpenMP and is outlined into a function that will be passed to each thread in the team. OpenMP requires that a structured block has one point of entry at the top and one point of exit at the bottom (the only exception being a statement that shuts down the entire program).

As with multithreaded programming environments in general, OpenMP is a shared memory API. The threads "belong" to a single process and they all share the heap associated with the process. Variables visible outside a parallel construct are by default shared inside the construct. Variables created inside a construct are by default private to the construct (i.e., there is a copy of the variable for each thread in the team). It is good form in OpenMP programming to make the status of variables explicit in an OpenMP construct which we do with the shared and private clauses in lines 14 and 15 in figure 1.

In an SPMD program, you need to find the rank (or thread number) and number of threads. We do this with OpenMP runtime

```
1  from numba import njit
2  import numpy as np
3  from numba.openmp import openmp_context as openmp
4  from numba.openmp import omp_get_wtime, omp_get_thread_num
5  from numba.openmp import omp_get_num_threads
6
7  MaxTHREADS = 32
8
9  @njit
10 def piFunc(NumSteps):
11     step = 1.0/NumSteps
12     partialSums = np.zeros(MaxTHREADS)
13     startTime = omp_get_wtime()
14     with openmp('''parallel shared(partialSums,numThrds)
15                    private(threadID,i,x,localSum)'''):
16        threadID = omp_get_thread_num()
17        with openmp("single"):
18           numThrds = omp_get_num_threads()
19           print("number of threads =",numThrds)
20
21        localSum = 0.0
22        for i in range(threadID,NumSteps,numThrds):
23            x = (i+0.5)*step
24            localSum = localSum + 4.0/(1.0 + x*x)
25        partialSums[threadID] = localSum
26
27     pi = step*np.sum(partialSums)
28     runTime = omp_get_wtime() - startTime
29     print(pi, NumSteps, runTime)
30     return pi
31
32 start = omp_get_wtime()
33 pi = piFunc(100000000)
34 print("time including JIT time",omp_get_wtime()-start)
```

Fig. 1: *A program using the SPMD pattern to numerically approximate a definite integral that should equal pi*

functions in lines 16 and 18. The rank of a thread, threadID, is private since each thread needs its own value for its ID. All threads in a single team, however, see the same value for the number of threads (numThrds) so this is a shared variable. In multithreaded programming, it is a data race if multiple threads write to the same variable; even if the value being written is the same for each thread. So we must assure that only one thread sets the value for the number of threads. This is done with the single construct on line 17.

The extent of the parallel algorithm is the for-loop starting at line 22. Each thread starts with a loop iteration (i) equal to its rank, which is incremented by the number of threads. The result is loop iterations dealt out as if from a deck of cards. This commonly used technique is called a "cyclic distribution of loop iterations". This loop is summing values of the integrand which we accumulate into a private variable for each thread. Since we need to later combine these local sums to get the final answer (on line 27), we store the local sum into a shared array (partialSums) on line 25.

The parallel region ends at line 25 at which point the team of threads join back together and the single original thread continues. We time the block of code with calls to the OpenMP runtime function omp_get_wtime() (lines 13 and 28) which returns the elapsed time since a fixed point in the past. Hence, differences in time values returned from omp_get_wtime() provides the elapsed time for execution of a block of code. We show runtimes for this SPMD program in figure 2 and compare to the same algorithm implemented in C. The runtimes are comparable. This supports our assertion that once the path for execution passes from the Python interpreter to machine code via Numba and LLVM, performance should match that from lower-level programming languages passing through the same LLVM/runtime infrastructure.

N	SPMD	loop	task	C SPMD	C loop	C Task
1	0.450	0.447	0.453	0.448	0.444	0.445
2	0.255	0.252	0.245	0.242	0.245	0.222
4	0.164	0.160	0.146	0.149	0.149	0.131
8	0.0890	0.0890	0.0898	0.0826	0.0827	0.0720
16	0.0503	0.0520	0.0517	0.0451	0.0451	0.0431

Fig. 2: *Programs to approximate a definite integral whose value equals pi using the SPMD, loop level, and divide-and-conquer/task pattern. Runtimes in seconds for PyOMP and analogous C programs. Programs were run on an Intel(R) Xeon(R) E5-2699 v3 CPU with 18 cores running at 2.30 GHz. For the C programs we used the Intel(R) icc compiler version 19.1.3.304 as "icc -qnextgen -O3 -fiopenmp".*

Loop Level Parallelism

The Loop Level Parallelism pattern is where most people start with OpenMP. This is shown in figure 2. The code is almost identical to the serial version of the program. Other than the import and timing statements, parallelism is introduced through a single with statement to express the parallel for construct. This construct creates a team of threads and then distributes the iterations of the loop among the threads. To accumulate the summation across loop iterations, we include the reduction clause. This clause defines reduction with the + operator over the variable sum. A copy of this variable is created for each thread in the team. It is initialized to the identity for the operator (which in this case is zero). At the end of the loop, all the threads wait for the other threads (a synchronization operation called a barrier). Before exiting the barrier, the local copies of sum are combined into a single value, that value is combined with the value of sum from before the parallel loop construct, and the threads join so only the single, original thread continues.

This program uses the default number of threads established outside the code of the program. This is set using an environment variable, OMP_NUM_THREADS; hence, we run our program pi_loop with the command line:

OMP_NUM_THREADS=16 python pi_loop.py

The results for this pattern are shown as the second column in figure 2. Once again, the performance is similar to that achieved with the C version of the program.

Tasks and Divide and Conquer

Our final pattern is more complex than the other two. This important pattern is heavily used by more advanced parallel programmers. A wide range of problems including optimization problems, spectral methods, and cache oblivious algorithms use the Divide and Conquer pattern. The general idea is to define three basic phases of the algorithm: split, compute, and merge. The split phase recursively divides a problem into smaller subproblems. After enough splits, the subproblems are small enough to directly compute in the compute phase. The final phase merges subproblems together to produce the final answer.

A Divide and Conquer solution to our pi problem is shown in figure 3. We start by creating a team of threads on line 37. We use the single construct to select one thread to start the algorithm with a call to our recursive function piComp(). With the single construct, one thread does the computation within the construct while the other threads wait at the end of the single construct (a

```
1  from numba import njit
2  from numba.openmp import openmp_context as openmp
3  from numba.openmp import omp_get_wtime
4
5  @njit
6  def piFunc(NumSteps):
7      step = 1.0/NumSteps
8      sum = 0.0
9      startTime = omp_get_wtime()
10
11     with openmp ("parallel for private(x) reduction(+:sum)"):
12         for i in range(NumSteps):
13             x = (i+0.5)*step
14             sum += 4.0/(1.0 + x*x)
15
16     pi = step*sum
17     runTime = omp_get_wtime() - startTime
18     print(pi, NumSteps, runTime)
19     return pi
20
21 start = omp_get_wtime()
22 pi = piFunc(100000000)
23 print("time including JIT time",omp_get_wtime()-start)
```

Fig. 3: *A program using the Loop Level Parallelism pattern to numerically approximate a definite integral that should equal pi*

```
1  from numba import njit
2  from numba.openmp import openmp_context as openmp
3  from numba.openmp import omp_get_wtime
4  from numba.openmp import omp_get_num_threads
5  from numba.openmp import omp_set_num_threads
6
7  NSTEPS  = 100000000
8  MIN_BLK = 1024*256
9
10 @njit
11 def piComp(Nstart, Nfinish, step):
12     iblk = Nfinish-Nstart
13     if(iblk<MIN_BLK):
14         sum = 0.0
15         for i in range(Nstart,Nfinish):
16             x= (i+0.5)*step
17             sum += 4.0/(1.0 + x*x)
18     else:
19         sum1 = 0.0
20         sum2 = 0.0
21         with openmp ("task shared(sum1)"):
22             sum1 = piComp(Nstart, Nfinish-iblk/2,step)
23
24         with openmp ("task shared(sum2)"):
25             sum2 = piComp(Nfinish-iblk/2,Nfinish,step)
26
27         with openmp ("taskwait"):
28             sum = sum1 + sum2
29
30     return sum
31
32 @njit
33 def piFunc(NumSteps):
34     step = 1.0/NumSteps
35     sum = 0.0
36     startTime = omp_get_wtime()
37     with openmp ("parallel"):
38         with openmp ("single"):
39              sum = piComp(0,NumSteps,step)
40
41     pi = step*sum
42     runTime = omp_get_wtime() - startTime
43     print(" pi = ",pi," in ",runTime," seconds")
44     return pi
45
46 startWithJit = omp_get_wtime()
47 pi = piFunc(NSTEPS)
48 print(" time with JIT = ",omp_get_wtime()-startWithJit)
```

Fig. 4: *A program using the Divide and Conquer pattern with tasks to numerically approximate a definite integral that should equal pi.*

so-called implied barrier). While those threads wait at the barrier, they are available for other computation on behalf of the program.

Inside the piComp() function, we test if the problem size is small enough for direct computation (is it smaller than a minimum block size) on line 13. If it is, we just compute the numerical integration for that block of loop iterations (lines14 to 17) and return the partial sum (line 30). If an instance of the function, piComp(), has a block of iterations greater than MIN_BLK, we enter the split phase of the algorithm. The split occurs in lines 19 to 25 using the task construct. This construct takes the code associated with the construct (in this case, a single line) and outlines it with its data environment to define a task. This task is placed in a queue for other threads in the team to execute. In this case, that would be the threads waiting at the barrier defined with the single construct on line 38.

As tasks complete, we enter the merge phase of the algorithm. This occurs at lines 27 and 28. The task that launches a pair of tasks must wait until its "child tasks" complete. Once they do, it takes the results (the shared variables sum1 and sum2), combines them, and returns the result. The results are summarized in figure 2. Even though the code is more complex than for the other two patterns, the runtimes for this simple problem are comparable to the other patterns for both Python and C.

Numba and the implementation of PyOMP

Numba is a Just In Time (JIT) compiler that translates Python functions into native code optimized for a particular target. The Numba JIT compiles PyOMP to native code in 4 basic phases.

- Untyped phase: Numba converts Python bytecode into its own intermediate representation (IR), including "with" contexts that are OpenMP-represented in the IR as "with" node types, and performs various optimizations on the IR. Later, Numba removes these "with" nodes by translating them to other node types in the IR. For our PyOmp implementation, we added a new OpenMP node type into the IR, and we convert OpenMP with contexts into these new OpenMP IR nodes.
- Type inference phase: Numba performs type inference on the IR starting from the known argument types to the function and then performs additional optimizations. No changes were made to the Numba typed compilation phase to support OpenMP.
- IR conversion phase: Numba converts its own IR into LLVM IR.
- Compilation phase: Numba uses LLVM to compile the LLVM IR into machine code and dynamically loads the result into the running application.

For PyOmp, we replaced the mainline LLVM normally used by Numba with the custom LLVM used within the Intel compiler, icx. This custom icx LLVM supports the bulk of OpenMP through two special function calls to demarcate the beginning and end of OpenMP regions (we will refer to these as OpenMP_start and OpenMP_end respectively) and LLVM tags on those function calls are used to apply the equivalent of OpenMP directives/clauses to those regions. Our PyOMP prototype passes the equivalent of the "-fiopenmp" icx compiler option to the icx LLVM which causes it to convert the demarcated OpenMP regions into OpenMP runtime function calls. The Intel OpenMP runtime is thus also needed and loaded into the process by the PyOMP prototype OpenMP

OpenMP construct, function, or clause	Explanation
with openmp("parallel"):	Create a team of threads that execute a parallel region
with openmp("for"): with openmp("parallel for"):	Used inside a parallel region to split up loop iterations among the threads. A combined construct equivalent to a parallel construct followed by a for.
with openmp ("single"):	One thread does the work while the others wait for it to finish
with openmp("task"): with openmp("taskwait"):	Create an explicit task for deferred execution of work within the construct. Wait for all tasks in the current task to complete.
with openmp("barrier"): with openmp("critical"):	All threads arrive at a barrier before any proceed. Only one thread at a time can execute the code inside a critical region
reduction(op:list)	A clause used with for to combine values with op across all the threads
schedule(static [,chunk])	A clause used with for to control how loop iterations are scheduled onto threads
private(list) firstprivate(list) shared(list)	A clause on parallel, for, or task. Create a private copy of the variables in the list. A clause on parallel, for, or task. Private but the variables equal original value. A clause on parallel, for, or task. Make the variables in the list shared
default(none)	A clause that forces explicit definition of variables as private or shared.
omp_get_num_threads() omp_get_thread_num() omp_set_num_threads(int)	Return the number of threads in a team Return an ID ranging from 0 to the number of threads minus one Set the number of threads to request for subsequent parallel regions
omp_get_wtime()	Call before and after a block of code to measure the elapsed time.
OMP_NUM_THREADS=N	Set the default number of threads until reset by omp_set_num_threads()

Fig. 5: *Summary of the elements of OpenMP included in PyOMP. This includes constructs (using the Python with statement), clauses that modify constructs, functions from the OpenMP runtime library, and a single environment variable. These elements include 19 of the elements in the OpenMP Common core (missing only nowait and the dynamic schedule).*

system. In PyOMP during the third phase, we convert the Numba OpenMP IR nodes to these two special function calls along with the corresponding LLVM tags. Additional details are described later.

OpenMP includes a number of runtime functions to interact with the system as a program runs. This is used to manage the number of threads, discover thread IDs, measure elapsed time, and other operations that can only occur as a program executes. For these functions, our prototype using CFFI to make those functions from the OpenMP runtime accessible from Python. The importing of some of these functions such as omp_get_num_threads, omp_get_thread_num, omp_get_wtime, and omp_set_num_threads can be seen, for example, in the initial "from numba.openmp import" ... lines at the beginning of the code example in figure 1.

Converting PyOMP with clauses to Numba IR

When removing OpenMP with contexts and replacing them with OpenMP IR nodes, Numba provides basic block information to demarcate the region that the with context covers. PyOMP places one OpenMP IR node at the beginning of this region and one at the end with a reference from the end node back to the start node to associate the two. To determine what to store in the OpenMP IR node, PyOMP first parses the string passed to the OpenMP with context to create a parse tree. Then, we perform a postorder traversal of the parse tree, accumulating the information as we go up the tree until we reach a node that has a direct OpenMP LLVM tag equivalent. At this point, we convert the information from the sub-tree into tag form and then subsequently pass that tag up the parse tree. These tags are accumulated as lists of tags up the parse tree until the traversal reaches a top-level OpenMP construct or directive, which have their own tags. Some of these directives are simple and require no additional processing whereas others, particularly those that support data clauses, require additional clauses to be added to the Numba OpenMP node that are not necessarily explicitly present in the programmer's OpenMP string. For example, all variables used within the parallel, for and parallel for directives must be present as an LLVM tag even if they are not explicitly mentioned in the programmer's OpenMP statement. Therefore, for these directives our PyOmp prototype performs a use-def analysis of the variables used within the OpenMP region to determine if they are also used before or after the OpenMP region. If they are used exclusively within the OpenMP region then their default data clause is private. In all other cases, the default data clause is shared but of course these defaults can be overridden by explicit data clauses in the programmer OpenMP string. For looping constructs, icx LLVM only supports loops in a certain canonical form that differs from the standard Numba IR loop form. For this purpose, our prototype transforms the Numba IR loop structure to match the icx LLVM loop structure.

Converting PyOMP Numba IR to LLVM

When a Numba OpenMP IR node is encountered in the process of converting Numba IR to LLVM IR, that node is converted to an LLVM OpenMP_start (or OpenMP_end) call. Inside the Numba OpenMP node is a list of the clauses that apply to this OpenMP region and we perform a 1-to-1 conversion of that list of clauses into a list of LLVM tags on the LLVM OpenMP_start call. We emit code that captures the result of the LLVM OpenMP_start call and we pass that result as a parameter to the OpenMP_end, which allows LLVM to match the beginning and end of OpenMP regions.

In the process of converting Numba OpenMP IR nodes and the intervening OpenMP regions to LLVM, we disable certain Numba functionality. Numba unifies the handling of exceptions and return values by adding an additional hidden parameter to functions it compiles that indicates whether the function has returned normally with a given return value or is propagating an exception. After a call site, Numba inserts code into a caller to check if the callee function is propagating an exception by inspecting the callee's hidden parameter. If there is an exception, the caller places that exception in its own hidden parameter and returns. However, this approach of using returns for exceptions breaks the icx LLVM

requirement that OpenMP regions be single-entry and single-exit. Likewise, exceptions generated from within the caller, such as divide-by-zero, also fill in the exception information in the hidden parameter and immediately return, again breaking the single-entry/exit requirement. It is not currently possible to explicitly catch such exceptions in PyOMP Numba regions because the Numba exception catching mechanism also generates control flow that violates single-exit. As such, in our PyOMP prototype, inside OpenMP regions, exception handling is currently elided.

The Numba process of converting Numba IR to LLVM IR introduces many temporary variables into the LLVM IR that are not present in the Numba IR. Thus, these variables are not visible in the untyped phase in which the data clauses for all variables accessed in OpenMP regions are determined. Such temporaries used solely within an OpenMP region should be classified as private in the tags associated with the surrounding OpenMP region's OpenMP_start demarcation function call. In PyOMP, we implemented a callback in the Numba function that creates these LLVM temporary variables such that we can learn of the existence of these new variables and to add them as private to the previously emitted tags of the surrounding OpenMP region.

Finally, certain OpenMP directives such as single and critical, require the use of memory fences with acquire, release, or acquire/release memory orders. Our prototype knows which directives require which kind of fences and we store that information in the Numba OpenMP IR node as those are created during the untyped phase. During conversion of those OpenMP IR nodes to LLVM, if the node require memory fences then we insert the equivalent LLVM fence instructions into the LLVM IR.

Results

The key result of this paper is that PyOMP works. As we saw in figure 2, we achieved reasonable speedups for the three key patterns that dominate OpenMP programming where by the word "reasonable" we mean "achieving performance similar to that from C". The pi programs, however, are "toy programs". They are useful pedagogically but are far removed from actual applications.

One step above a "toy program" is dense matrix multiplication. While this is a simple program lacking in the inevitable complexities faced by real applications, dense matrix multiplication uses a familiar loop-nest and data access patterns found in real applications. It has the further advantage that dense matrix multiplication over double precision values (DGEMM) is compiler-friendly. If a compilation tool-chain is going to work well, DGEMM is where this would be most apparent.

Our DGEMM code comes from the Parallel Research Kernels (PRK) [VdW14] version 2.17. All code is available from the PRK repository [PRK]. The PyOMP code is summarized in figure 6. The Numba JIT was done with the 'fastmath' option. This resulted in a 20% performance improvement. Numba and therefore PyOMP requires that any arrays use NumPy. They are allocated and initialized on lines 10 to 12 and then assigned values on lines 16 to 18 such that the matrix product is known and available for testing to verify correctness. The multiplication itself occurs on lines 21 to 25. The ikj loop order is used since it leads to a more cache-friendly memory access pattern. The elapsed time is found (dgemmTime) and reported as GF/s (billions of floating point operations per second or GFLOPS).

We compare performance from PyOMP to the analogous program written with C/OpenMP, NumPy arrays with the ikj loop-nest, and a call to the matrix multiplication function included with

```
1  from numba import njit
2  import numpy as np
3  from numba.openmp import openmp_context as openmp
4  from numba.openmp import omp_get_wtime
5
6  @njit(fastmath=True)
7  def dgemm(iterations,order):
8
9      # allocate and initialize arrays
10     A = np.zeros((order,order))
11     B = np.zeros((order,order))
12     C = np.zeros((order,order))
13
14     # Assign values to A and B such that
15     # the product matrix has a known value.
16     for i in range(order):
17         A[:,i] = float(i)
18         B[:,i] = float(i)
19
20     tInit = omp_get_wtime()
21     with openmp("parallel for private(j,k)"):
22           for i in range(order):
23               for k in range(order):
24                   for j in range(order):
25                       C[i][j] += A[i][k] * B[k][j]
26
27     dgemmTime = omp_get_wtime() - tInit
28
29     # Check result
30     checksum = 0.0;
31     for i in range(order):
32         for j in range(order):
33             checksum += C[i][j];
34
35     ref_checksum = order*order*order
36     ref_checksum *= 0.25*(order-1.0)*(order-1.0)
37     eps=1.e-8
38     if abs((checksum - ref_checksum)/ref_checksum) < eps:
39         print('Solution validates')
40         nflops = 2.0*order*order*order
41         print('Rate (MF/s): ',1.e-6*nflops/dgemmTime)
42     else:
43         print('ERROR: ', checksum, ref_checksum,'\n')
```

Fig. 6: *A PyOMP program to multiply two matrices.*

NumPy. Code fragments for these cases are shown in figure 7. The C DGEMM program was compiled with the Intel(R) icc compiler version 19.1.3.304. The compiler command line was:

```
icc -std=c11 -pthread -O3 -xHOST  -qopenmp
```

We ran all computations on an an Intel(R) Xeon(R) E5-2699 v3 CPU CPU with 18 cores running at 2.30 GHz. For the multi-threaded programs with OpenMP we forced the threads to map onto specific cores with one thread per core using the following pair of environment variables:

```
export OMP_PLACES="{0},{1},{2},{3},{4}"
export OMP_PROC_BIND=close
```

where the numbers in OMP_PLACES continued up to the number of threads used in the computation. When combined with the processor binding term (close) this connected the OpenMP thread ID with the core ID (e.g., OpenMP thread ID 0 ran on core 0). This way, we knew that the C and Python OpenMP programs used precisely the same cores and had the same relationship to the memory controllers on the chip.

We choose a matrix order large enough to create sufficient work to overcome memory movement and thread overhead. These matrices were too large for the computation to complete on our system for matrices represented through Python lists. Using NumPy arrays with triply nested loops in i,k,j order, the com-

```
1  \\ The computation from the C PRK DGEMM
2   #pragma omp parallel for
3      for (jg=0; jg<order; jg++)
4      for (kg=0; kg<order; kg++)
5      for (ig=0; ig<order; ig++)
6        C_arr(ig,jg) += A_arr(ig,kg)*B_arr(kg,jg);
7
8  \\--------------------------------------------
9
10    # The python program for DGEMM using NumPy arrays
11    # 0.00199 GFLOPS for order 1000 matrices (one thread)
12       for i in range(order):
13           for k in range(order):
14               for j in range(order):
15                   C[i][j] += A[i][k] * B[k][j]
16
17 \\--------------------------------------------
18
19    # The python program for DGEMM using the matmul
20    # function from NumPy
21    # 9.02 GFLOPS order 1000 matrix
22    C += numpy.matmul(A,B)
```

Fig. 7: *We compare our PyOMP program to three other cases: C with OpenMP, serial code using the NumPy arrays, and the 'matmul()' built in function for matrix multiplication. All programs use the same matrices, tests for correctness, and performance metrics (shown in figure 6), hence that code is not reproduced here.*

threads	PyOMP	C with OpenMP
1	3.416 ± 0.017	3.506 ± 0.035
2	6.718 ± 0.068	6.902 ± 0.078
4	12.33 ± 0.166	12.67 ± 0.184
8	21.79 ± 0.226	22.33 ± 0.381
16	41.70 ± 1.130	42.94 ± 1.194

Fig. 8: *The PyOMP and the C are comparable with the C results consistently around 2.8 percent faster than the results from PyOMP. We performed a Welch's T-test for the two sets of data. The test showed that while the difference between the PyOMP and C cases are small, they are statistically significant to the 99% confidence level.*

putation ran at 0.00199 GFLOPS. For our scalability studies, all runs were repeated 250 times. Averages and standard deviations in GFLOPS are reported. Results are shown in figure 8. For the PyOMP results, we do not include the JIT times. These were only done once per run (i.e. not once per iteration) and took on the order of two seconds.

Parallel Research Kernel DGEMM gigaFLOPS per second for order 1000 matrices. Results are the average and standard deviation of 250 runs for execution on an Intel(R) Xeon(R) E5-2699 v3 CPU with 18 cores running at 2.30 GHz. The python results do not include the time to JIT compile the python code. This one-time cost was observed to add around 2 seconds to the runtime.

If we use NumPy and call the matrix multiplication function provided with NumPy (line 22 in figure 7, the order 1000 DGEMM ran at 11.29 +/- 0.58 GFLOPS with one thread (using the matmul() function from NumPy). This high performance serves to emphasize that while DGEMM is a useful benchmark to compare different approaches to writing code, if you ever need to multiply matrices in a real application, you should use code in a library produced by performance optimization experts.

Discussion

In the paper "There's plenty of room at the top..." [Lei20], much was made of the low performance available from code written in Python. They motivated their discussion using DGEMM. The implication was that when you care about performance, rewrite your code in C. We understand that sentiment and often use that strategy ourselves. Our goal, however, is to meet programmers "on their turf" and let them "stay with Python".

One of the key challenges to the "stay with Python" goal is multithreading. Because of the GIL, if you want multithreaded code to execute in parallel, you can't use Python. In this paper, we have addressed this issue by using Numba to map onto LLVM and the OpenMP hooks contained therein. This resulted in our Python OpenMP system called PyOMP.

The performance from PyOMP was within a few percent of performance from OpenMP code written in C. Performance differences were statistically significant, but we believe not large enough to justify rewriting code in C. This holds for a subset of OpenMP supported in PyOMP (known as the "Common Core" [Mat19]) and for the three fundamental design patterns used by OpenMP programmers.

PyOMP is a research prototype system. It is a proof-of-concept system we created to validate that Numba together with LLVM could enable multithreaded programming in Python through OpenMP. A great deal of work is needed to move from a research prototype to a production-ready tool for application programmers.

- We need to develop a formal test suite. We currently have a test suite that covers each PyOMP OpenMP construct in isolation. In those tests, we use a very limited subset (e.g., ints, floats, NumPy arrays, prints, assignments) of the Python features supported by Numba [Numba]. We need a test suite that covers the combinations of OpenMP constructs encountered in real OpenMP applications with the full set of data types and Python features supported by Numba. In this process, we will note Numba features incompatible with OpenMP (such as ParallelAccelerator [And17]); fixing the cases we can fix and documenting those we can't.
- We need to work out the details for how we will distribute this code. We used the Intel production LLVM-based compiler which ties PyOMP to Intel proprietary tools. We need to investigate whether the OpenMP support in the Intel open source release of LLVM is sufficient to support PyOMP.
- Currently, exception handling in PyOMP is disabled due to the interaction of how Numba manages exceptions with how LLVM manages execution of structured blocks in OpenMP. We are investigating ways to address this problem, but don't have a solution at this time.
- We currently disable the Numba static single assignment mode (SSA). In this mode, Numba creates variants of variables. Those names are difficult to track relative to the data environment of OpenMP. We believe we can account for these variants in PyOMP, but we have not done so yet.

In additions to refinement to PyOMP itself, we need to conduct a formal benchmarking effort with benchmarks that exercise the system in the way real applications would. In this effort we also need to compare to the performance of other systems for parallel programming for a CPU with Python. In particular, we want to understand the performance tradeoffs between PyOMP, Dask, MPI4Py, and implicit parallelism through Numba's ParallelAccelerator.

REFERENCES

[And17] T. Anderson, H. Liu. L. Kuper, E. Totoni, J. Vitek and T. Shpeisman. "Parallelizing Julia with a Non-Invasive DSL" 31st European Conference on Object-Oriented Programming (ECOOP 2017), Leibniz International Proceedings in Informatics (LIPIcs)}, vol. 74, pp. 4.1-4.29, 2017.

[deS18] B. de Supinski, T. Scogland, A. Duran, M. Klemm, S. Bellido, S. Olivier, C. Terboven, T. Mattson. "The Ongoing Evolution of OpenMP", Proceedings of the IEEE, Vol 106, No. 11, 2018

[Lam15] S. Lam, K. Siu, A. Pitrou, and S. Seibert. "Numba: A llvm-based python jit compiler:, Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC}, pp. 1-6, 2015.

[Lei20] C. Leiserson, N. Thompson, J. Emer, B. Kuszmaul, B. Lampson, D. Sanchez, and T. Schardl, "There's plenty of room at the Top: What will drive computer performance after Moore's law?", Science, Vol. 368, P. 6495, 2020.

[Mat19] T. Mattson, Y. He, and A. Koniges. "The OpenMP Common Core: Making OpenMP Simple Again", MIT Press, 2019.

[Numba] "Python features supported by Numba", https://numba.pydata.org/numba-doc/dev/reference/pysupported.html.

[NPar] "Automatic parallelization with jit", https://numba.pydata.org/numba-doc/latest/user/parallel.html, 2021.

[PRK] "Parallel Research Kernels repository", https://github.com/ParRes/Kernels

[PyP21] "PYPL PopularitY of Programming Language", https://pypl.github.io/PYPL.html, collected May, 2021.

[VdW14] R. van der Wijngaart and T. Mattson, "The Parallel Research Kernels: A tool for architecture and programming system investigation" IEEE High Performance Extreme Computing, 2014.

CNN Based ToF Image Processing

Marian-Leontin Pop[‡], Szilard Molnar[‡], Alexandru Pop[‡], Benjamin Kelenyi[‡], Levente Tamas[‡*], Andrei Cozma[§]

https://www.youtube.com/watch?v=kANXhHwFrCo

Abstract—In this paper a Time of Flight (ToF) camera specific data processing pipeline is presented, followed by real life applications using artificial intelligence. These applications include use cases such as gesture recognition, movement direction estimation or physical exercises monitoring. The whole pipeline for the body pose estimation is described in details, starting from generating and training phases to the pose estimation and deployment. The final deployment targets were Nvidia Xavier NX and AGX platforms receiving data from an Analog Devices ToF camera.

Index Terms—transfer learning, ToF, python

Introduction

In recent years the evolution of deep neural networks has affected the way in which Time of Flight (ToF) images are processed. Images from ToF cameras are usually obtained as synchronized depth and infrared (IR) image pairs. The customization of the existing deep nets to the IR and depth images allows us to reuse the existing models and techniques from this emerging domain. The applications targeted are ranging from person detection, counting, activity analysis to volumetric measurements, mapping and navigation with mobile agents. In the following parts the introduction to the specific ToF imaging, custom data processing and CNN based solutions are presented [TC21]. Although for the 2D data a bunch of CNN based solutions exists, for the 3D data [GZWY20] only some base architectures were widespread such as Pointnet [QSMG17], while for the calibration between different sensing modalities can be done in an efficient way according to [FTK19].

ToF specific imaging

The 2D image processing part is a customized IR image module based on transfer learning for bounding box estimation, skeleton extraction and hardware specific model translation. The latter is relevant in order to have a light-weight embedded solution running on limited floating-point precision hardware platforms such as Jetson Nvidia Family. As the existing CNN models are mainly with the focus on colour images, thus ones has to adopt transfer learning as a method to finetune the existing CNN models such as VGG, MobileNet for the infrared or depth images specific to ToF cameras. This solution seemed to be effective in terms of precision

‡ *Technical University of Cluj-Napoca*
∗ *Corresponding author: Levente.Tamas@aut.utcluj.ro*
§ *Analog Devices International*

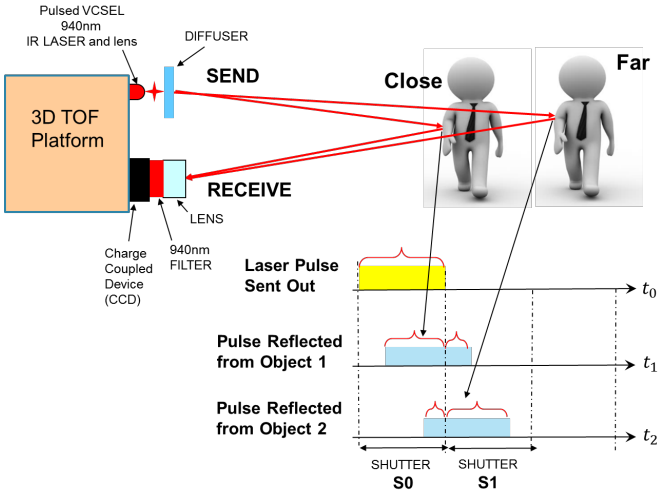

Fig. 1: *Exemplification of ToF camera*

and runtime on embedded devices (e.g Jetson NX or AGX). For the skeleton detection part we relied on the real-time Tensorflow optimized module for the Jetson product family, however for the generic GPU enabled devices we had to tailor our models since these are custom solutions.

Custom pipeline for ToF data

The main role of the depth image preprocessing part is the filtering and bounding box estimation for the 3D ROI. The filtering is essential for the embedded device in order to reduce the computational overload. For the filtering pipeline we considered three interconnected filters: voxel, pass-through and outlier filter as this is visible in Figure 2. All these implementations are open source library based variants. The details of the filtering were reported in [TC21].

Low level ToF image pre-processing - ToFNest

In ToFNest we are approximating surface normals from depth images, recorded with Time-of-Flight cameras. The approximation is done using a neural network. The base of our neural network is the PyTorch library, since the whole process is done using Python 3.6 as our programming language. Using PyTorch we have created a Feature Pyramid Network type model ([LDG+17]).

The main pipeline of the data was the following: first we read the depth images with OpenCV (alongside the depth information we could also use the infrared information or the rgb information from the camera as well, thus adding more information to work

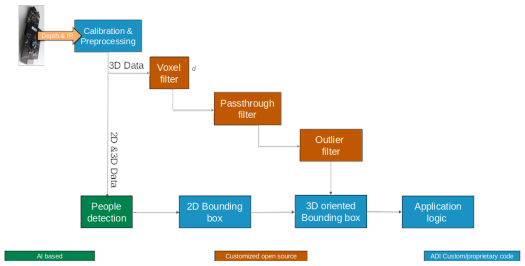

Fig. 2: *Processing pipeline for ToF camera*

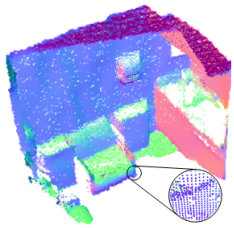

Fig. 3: *Exemplification of ToF normal estimation*

with), then we prepare them with numpy. From a numpy array it is easy to convert it to a torch tensor on the GPU, which then creates the predictions about the surface normals. An example of the prediction can be seen in Figure 3, where the direction of the normal vectors are decoded with RGB images.

The following code represents the loss:

```
pred=pred*2-1
gt=gt*2-1
inner_product = (pred * gt).sum(dim=1).unsqueeze(1)
cos = inner_product / 2
angle = torch.acos(cos)
if not args.orient_normals:
    angle[angle>1.57]=3.14-angle[angle>1.57]
loss = torch.mean(angle)
return loss
```

The results were accurate relative to other techniques, but the time was much less. The time being less means that at least 100 times faster. This can be due to the fact, that this method works with images, instead of point clouds as other methods do. This makes it much faster, as this was reported in [MKT21].

Our method was evaluated by verifying only the angles between the lines, not the exact directions of the vectors (this was the case in the other methods as well), but we can train that, although the results are going to get worse.

Furthermore, in order to get a real-time visualization about the predictions, we used rospy to read the images from ROS topics, and also to publish the normal estimation values to another ROS topic, that we could visualize using Rviz. This can be seen in the demo video.

Low level ToF image pre-processing - ToFSmooth

This whole pipeline and network, with some minor modifications can be also used to smoothen the depth image, thus making the point cloud smoother as well.

For the dataset we added gaussian noise of 5 and 10 cm to the original data, while we smoothed the original data with PointCloudDenoising ([PFVM20]) method.

Our method got pretty close to the ground truth value, in most of the cases. Although, in the case of the original (originally

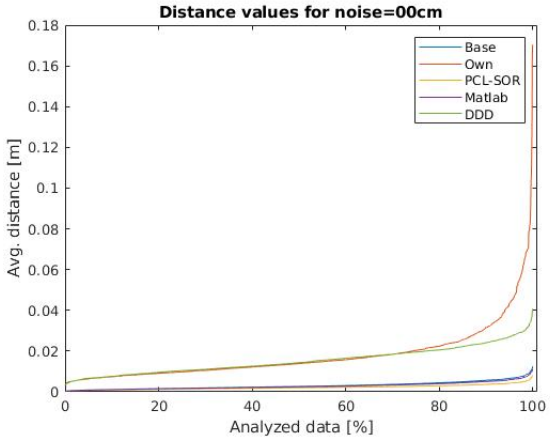

Fig. 4: *The average error for the original data*

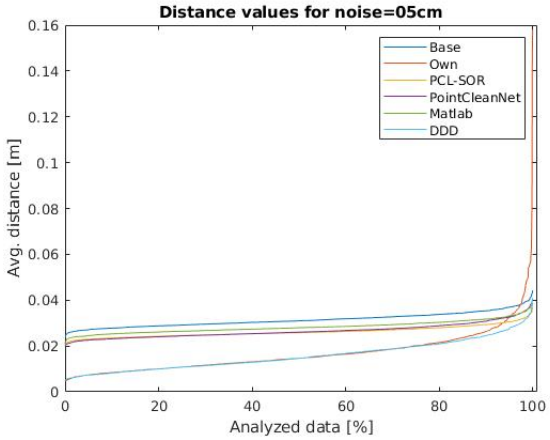

Fig. 5: *The average error for data with 5 cm gaussian noise*

fairly smooth) data resulted slightly worse results, then some other methods (for instance the PointCloud Library [RC11]), when we tested the smoothing for much more noisy data, our results barely changed, while other methods were highly compromised. A comparison between these cases can be seen in the next image 3 images:

Here we can see that our method kept very much the same

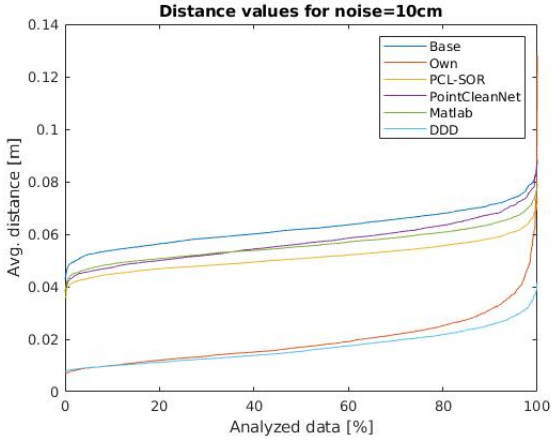

Fig. 6: *The average error for data with 10 cm gaussian noise*

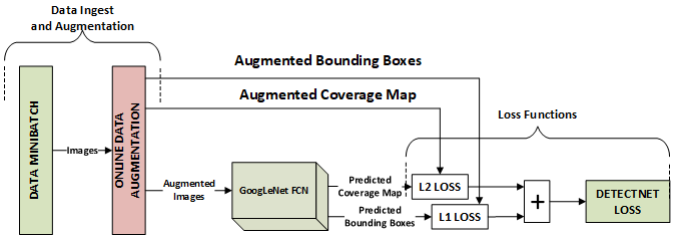

Fig. 7: *DetectNet structure for training*

throughout all the cases same as DeepDepthDenoising method ([SSC+19]), which is the only other method that we have found, that works with depth images as well, making it about the same as ours, but a little bit more polished. Also this method performs at the same speed as ours.

The jump in the error at the end of the scale is due to some denormalization bias that we need to fine-tune.

CNN based solutions

In this part we describe in details the person detection, action recognition and volumetric estimation applications.

Person detection from IR imaging

DetectNet is a detection algorithm based on the jetson-inference repository with people detection focus presented in [LLW+16] or [XLCH16]. This repository uses NVIDIA TensorRT for efficient implementation of neural networks on the Jetson platform, improving performance and energy efficiency through graphical optimizations, kernel fusion and FP16/INT8 accuracy.

Object detection requires a lot of information for training. DetectNet uses a large dataset, and each image contains multiple objects. For each object in the image, the trained model must detect both the object and the corner coordinates of the bounding box. Since the number of objects can vary in the training image set, it would be difficult to define the loss function if we choose the label format with variable length and dimensionality. This problem has been solved by introducing a 3-dimensional label format that enables DetectNet to ingest images of any size with a variable number of objects present.

In the Figure 7 you can see the architecture for the training process, which is based on 3 important steps:

- data layers ingest the training images and labels
- a fully-convolutional network (FCN) performs feature extraction and prediction of object classes and bounding boxes per grid square
- loss functions simultaneously measure the error in the two tasks of predicting the object coverage and object bounding box corners per grid square

In the final layers of DetectNet the openCV groupRectangles algorithm is used to cluster and filter the set of bounding boxes generated for grid squares with predicted coverage values greater than or equal to gridbox_cvg_threshold, which is specified in the DetectNet model definition prototxt file.

DetectNet also uses the “Python Layers” interface to calculate and output a simplified mean Average Precision (mAP) score for the final set of output bounding boxes. For each predicted bounding box and each ground truth bounding box the Intersection over Union (IoU) score is computed. IoU is the ratio of the overlapping areas of two bounding boxes to the sum of their areas.

Fig. 8: *Exemplification of people detection on infrared images based detection*

The pre-trained model accepts 3 channel images – RGB, by modifying the existing model, we have managed to detect and track people on the infrared image – 1 channel. With the help of the OpenCV library and the 3.7 Python programming language version, we have developed a script that modifies the contrast of the IR image; thus, we obtained a much better result than if we had not used this approach. This result can be seen in the Figure 8, where we can see that the people are detected on the IR image with high confidence.

To be able to run the algorithm in real-time we used the rospy client. With the help of this API, we have developed an efficient way to pass a ROS topic as input to our model. The algorithm was tested on a Jetson AGX, and the camera used was from Analog Devices (AD-96TOF1-EBZ). The result can be seen in the attached demo video.

Action recognition from IR images

This is a small tutorial for detecting the skeleton, or rather an approximation of the joints of a person, from an infrared image. In our setup we used one of the Analog Devices Time-of-Flight cameras, which provided us the infrared image, and an NVIDIA Jetson Xavier NX board, which is a compact system-on-module (SOM), very well suited for model inference.

As a baseline architecture model, we used the pretrained model from one of the NVIDIA-AI-IOT’s repositories: https://github.com/NVIDIA-AI-IOT/trt_pose . We used the TensorRT SDK in order to optimize our pretrained model for the Jetson Xavier NX platform, thus achieving a better performance in our model inference pipeline.

We also used, some of the Robot Operating System’s (ROS) tools for retrieving the camera infrared images and by using the rospy client library API we managed to transfer our infrared images to the network’s model. While this would have been an easy step using the CvBridge library, which provides an interface between ROS and OpenCV, this time was not the case, as we had some issues with this library. Because we are working on Jetson Xavier NX board, which comes with the latest OpenCV version, and CvBridge uses at its core an older version of OpenCv, we replaced the conversion from sensor_msgs/Image message type to the OpenCv image array made by CvBridge with a very useful numpy functionality which allowed us to make this conversion

Fig. 9: *Exemplification of skeleton detection on infrared images*

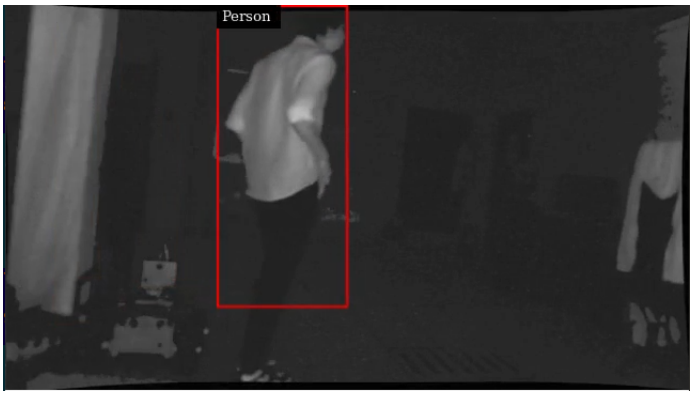

Fig. 10: *People detection algorithm running with the Deepstream SDK on the Jetson Xavier NX board*

flawlessly, while still achieving the same functionality and performance, because in fact, this was only a slight alteration of the underlying Python implementation of the CvBridge package. So, we replaced:

```
ir_image = CvBridge().imgmsg_to_cv2(image_msg,-1)
```

with:

```
ir_image = numpy.frombuffer(
image_msg.data,
dtype=numpy.uint8).reshape(
                        image_msg.height,
                        image_msg.width,
                        -1)
```

After making this conversion, we preprocessed the infrared image before feeding it to the neural network, using the OpenCv library. After this step we supply the model input with this preprocessed image, and we obtained the results which can be seen in the Figure 9.

Furthermore, as a side quest, because we tested the TensorRT SDK and we saw some good results in our model's inference, we decided to extend the infrared people detection application by integrating it with NVIDIA's Deepstream SDK. While this SDK brings further optimization to our model's inference performance and optimize the image flow along the inference pipeline by transferring the image on GPU for any kind of preprocessing required before it enters the model and even allowing us to serve multiple images, from multiple cameras, without a very drastic change in the model's inference speed. Even though these functionalities are important, we were interested by another functionality which the Deepstream SDK supports, this being the fact that is able to provide communication with a server and transmit the output of the neural network's model, which runs on the Jetson platform, to the server, for further data processing. This can be very useful in applications where we want to gather some sort of statistics or when our application has to make some decisions based on the output of our trained model, but we don't want to affect the Jetson's inference performance by overwhelming it with other processes. In the Figure 10, can be seen the result of the people detection algorithm made by using the Deepstream SDK, and below is the network's output received on our custom configured server when a person is detected:


```
{
"object" : {
"id" : "-1",
"speed" : 0.0,
"direction" : 0.0,
"orientation" : 0.0,
"person" : {
  "age" : 45,
  "gender" : "male",
  "hair" : "black",
  "cap" : "none",
  "apparel" : "formal",
  "confidence" : -0.10000000149011612
},
"bbox" : {
  "topleftx" : 147,
  "toplefty" : 16,
  "bottomrightx" : 305,
  "bottomrighty" : 343
},
"location" : {
  "lat" : 0.0,
  "lon" : 0.0,
  "alt" : 0.0
},
"coordinate" : {
  "x" : 0.0,
  "y" : 0.0,
  "z" : 0.0
}
}
```


Volumetric estimates for depth images

The goal of this research is to estimate the volume of objects using only depth images recorded with Time-of-Flight cameras. As a simplifying feature, we consider only box shaped objects, with clearly definable perpendicular planes. Two methods have been determined.The first method uses RANSAC algorithm to detect planes while the other one uses the ideas from [SSG+20].

The first algorithm iteratively finds the largest plane using RANSAC and uses euclidean extraction to remove it from the point cloud. Once the planes are determined and checked to see if they are perpendicular, the intersection lines of the planes are determined by projecting between them. The projections approximate a line and the points with the largest component difference determine the length of the line. This way iteratively the 3 intersecting line lengths can be determined once the planes are determined and checked for orthogonality.

An important observation is that it can compute the volume using 2 planes instead of 3. This is due to the fact that if 2 planes are orthogonal, the common line between them will be determined by 2 points that are also corner points for the object. By selecting a corner point and the two perpendicular planes, a third plane can be determined that is perpendicular to the other two and it contains

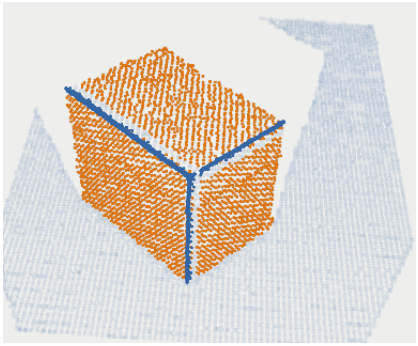

Fig. 11: *Planar detection*

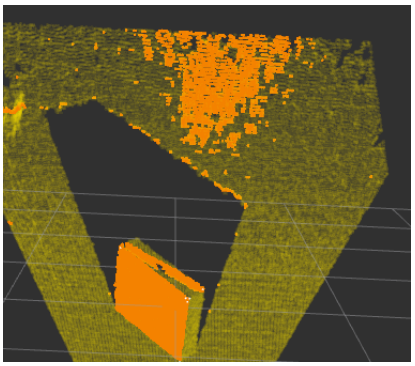

Fig. 12: *Limitations of planar segmentation*

the chosen point. Once the virtual third plane has been computed, the algorithm resumes as in the case with 3 determined planes.

An advantage of this method is that it uses readily available and studied functions for processing pointclouds. For a simple case of a box and floor plane, the algorithm accuracy depends on the level of noise the pointcloud has. The following code snippets ilustrate the functionality of the Planar Segmenting Volume computation method using 2 planes.

```
def volume_main(perp_thresh,min_nr_points,input_pcd)
floor=pcl_Planar_Ransac(input_pcd)
input_pcd=Euclidean_extraction(input_pcd)
if (pcl_Planar_Ransac(input_pcd)>min_nr_points)
   plane_1=Planar_Ransac(input_pcd)
input_pcd=Euclidean_extraction(input_pcd)
if(pcl_Planar_Ransac(input_pcd)>min_nr_points)
  plane_2=pcl_Planar_Ransac(input_pcd)
if(cos(plane_1 * plane_2)<perpendicular_threshold>)
  Volume=compute_volume_2_planes(plane1_plane2)
else
  (p_A, p_B)=line_points(plane_1,plane_2)
  plane_3=com_perp_plane_point(plane_1,plane_2,p_A)
  if(cos(plane_1*plane_3)<perpendicular_threshold>)
    Volume=compute_volume_2_planes(plane_2,plane_3)
```

```
def compute_volume_2_planes(plane_A,plane_B):
(p_AB_1, p_AB_2)=line_points(plane_A,plane_B)
plane_C=com_perp_plane_point(plane_A,plane_B,p_AB_1)
(p_AC_1,p_AC_2)=line_points(plane_A,plane_C)
(p_BC_1,p_BC_2)=line_points(plane_B,plane_C)
L1=distance(p_AB_1, p_AB_2)
L2=distance(p_AC_1, p_AC_2)
L3=distance(p_BC_1, p_BC_2)
Volume=L1*L2*L3
```

```
def line_points(plane_A,plane_B):
line_AB_pcd=pcl_project_inliers(plane_A,plane_B)
line_BA_pcd=pcl_project_inliers(plane_B,plane_A)
line_pcd=concat(line_AB_pcd,line_BA_pcd)
(abs_diff_x,p_AB_1_x,p_AB_2_x)=max_diff_x(line_pcd)
(abs_diff_y,p_AB_1_y,p_AB_2_y)=max_diff_y(line_pcd)
(abs_diff_x,p_AB_1_z,p_AB_2_z)=max_diff_z(line_pcd)
diff=max_diff(abs_diff_x,abs_diff_y,abs_diff_z)
(pointA, pointB)=points_max_diff(diff)
```

The downside of this method is that it can compute the volume only for one box. Noise and other objects in the scene can totally disrupt the volumetric estimate.

Due to these shortcomings, a new method for measuring the volume is studied, based on the work by [SSG+20]. Their paper, details an algorithm that uses pointclouds with normals computed in each point in order to determine collections of point pairs for which their normals satisfy the orthogonality constraint. The point pair collections will approximate the orthogonal planes. By determining the points contained by each orthogonal plane,

Fig. 13: *Corner detection*

projections can be made that approximate the intersecting lines of the orthogonal planes. By selecting the 3 lines that have the edge points closest to each other, volume of a box can be computed. The advantage of this method is that it allows the computation of the volume for multiple box shaped objects. The following code snippets show the usage of the Sommer's plane determination method to compute the volume.

```
def comp_vol_ortho(pcd,dmin,dman,votes,seg,thresh):
all_lines=sommer_planes(pcd,dmin,dman,votes,seg)
all_triplets=find_line_triplet(thresh,all_lines)
for i in all_triplets:
  line_1=distance(all_triplets[i][0])
  line_2=distance(all_triplets[i][1])
  line_3=distance(all_triplets[i][2])
  Volume[i]=line_1*line_2*line_3
```

```
def find_line_triplet(thresh):
for i in range(0,size(all_lines-3)):
 for j in range(i+1,size(all_lines-2)):
  for k in range(j+1,size(all_lines-1)):
   avr_p=(all_lines[i]+all_lines[j]+all_lines[k])/3
   if dist_each_to_avr(avr_p)<threshold:
      add_triplet(all_triplets)
```

Volume estimation using enhanced planar/corner detections was done using the training from [SSG+20]. The largest benefit of this method is that it does not rely on RANSAC and it can compute the volume for multiple objects.

This permits in further research to consider the idea of moving the camera in such a way to improve the volumetric measurement of multiple objects. This problem statement becomes equivalent to a Next Best View problem in which the view must optimize

the accuracy or availability of a volumetric measurement. This translates to the question "In which of the available positions does the camera need to be placed next in order to improve the volumetric measurement". A starting point for such an idea would be to use the neural network architecture used in [ZZL20], but change the loss function's objective from reconstruction to volumetric accuracy. By creating a scoring function for the volumetric accuracy, candidate new positions might be evaluated and chosen based on the input pointcloud.

Conclusion

In this paper we provided some guidelines for the ToF specific image processing using Python libraries. The demos are ranging from basic pointcloud processing to people detection and enhanced volume estimation.

Acknowledgement

The authors are thankful for the support of Analog Devices Romania, for the equipment list (cameras, embedded devices, GPUs) offered as support to this work. This work was financially supported by the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-III-P2-2.1-PTE-2019-0367. The authors are thankful for the generous donation from NVIDIA corporation for supporting this research.

REFERENCES

[FTK19] Robert Frohlich, Levente Tamas, and Zoltan Kato. Absolute pose estimation of central cameras using planar regions. *IEEE transactions on pattern analysis and machine intelligence*, 2019. doi:10.1109/TPAMI.2019.2931577.

[GZWY20] Abubakar Sulaiman Gezawa, Yan Zhang, Qicong Wang, and Lei Yunqi. A review on deep learning approaches for 3d data representations in retrieval and classifications. *IEEE Access*, 8:57566–57593, 2020. doi:10.1109/ACCESS.2020.2982196.

[LDG+17] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, 2017. doi:10.1109/CVPR.2017.106.

[LLW+16] Hengli Liu, Jun Luo, Peng Wu, Shaorong Xie, and Hengyu Li. People detection and tracking using rgb-d cameras for mobile robots. *International Journal of Advanced Robotic Systems*, 13(5):1729881416657746, 2016. doi:10.1177/1729881416657746.

[MKT21] Szilard Molnar, Benjamin Kelenyi, and Levente Tamas. Tofnest: Efficient normal estimation for time-of-flight depth cameras. In *Proc. of IROS 2021 (under review)*, 2021.

[PFVM20] Francesca Pistilli, Giulia Fracastoro, Diego Valsesia, and Enrico Magli. Learning graph-convolutional representations for point cloud denoising, 2020. arXiv:2007.02578.

[QSMG17] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017. arXiv:1612.00593, doi:10.1109/cvpr.2017.16.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 05 2011. doi:10.1109/ICRA.2011.5980567.

[SSC+19] Vladimiros Sterzentsenko, Leonidas Saroglou, Anargyros Chatzitofis, Spyridon Thermos, Nikolaos Zioulis, Alexandros Doumanoglou, Dimitrios Zarpalas, and Petros Daras. Self-supervised deep depth denoising. In *ICCV*, 2019. doi:10.1109/ICCV.2019.00133.

[SSG+20] Christiane Sommer, Yumin Sun, Leonidas Guibas, Daniel Cremers, and Tolga Birdal. From planes to corners: Multi-purpose primitive detection in unorganized 3d point clouds. *IEEE Robotics and Automation Letters (RA-L)*, 5(2):1764–1771, 2020. doi:10.1109/LRA.2020.2969936.

[TC21] Levente Tamas and Andrei Cozma. Embedded real-time people detection and tracking with time-of-flight camera. In *Proc. of SPIE Vol*, volume 11736, pages 117360B–1, 2021. doi:10.1117/12.2586057.

[XLCH16] Hongyang Xue, Yao Liu, Deng Cai, and Xiaofei He. Tracking people in rgbd videos using deep learning and motion clues. *Neurocomputing*, 204:70–76, 2016. Big Learning in Social Media Analytics. doi:10.1016/j.neucom.2015.06.112.

[ZZL20] Rui Zeng, Wang Zhao, and Yong-Jin Liu. Pc-nbv: A point cloud based deep network for efficient next best view planning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7050–7057. IEEE, 2020. doi:10.1109/iros45743.2020.9340916.

Cell Tracking in 3D using deep learning segmentations

Varun Kapoor‡*, Claudia Carabaña‡

Abstract—Live-cell imaging is a highly used technique to study cell migration and dynamics over time. Although many computational tools have been developed during the past years to automatically detect and track cells, they are optimized to detect cell nuclei with similar shapes and/or cells not clustering together. These existing tools are challenged when tracking fluorescently labelled membranes of cells due to cell's irregular shape, variability in size and dynamic movement across Z planes making it difficult to detect and track them. Here we introduce a detailed analysis pipeline to perform segmentation with accurate shape information, combined with BTrackmate, a customized codebase of popular ImageJ/Fiji software Trackmate, to perform cell tracking inside the tissue of interest. We developed VollSeg, a new segmentation method able to detect membrane-labelled cells with low signal-to-noise ratio and dense packing. Finally, we also created an interface in Napari, an Euler angle based viewer, to visualize the tracks along a chosen view making it possible to follow a cell along the plane of motion. Importantly, we provide a detailed protocol to implement this pipeline in a new dataset, together with the required Jupyter notebooks. Our codes are open source available at [Git].

Index Terms—3D segmentation, cell tracking, deep learning, irregular shaped cells, fluorescent microscopy.

Introduction

Live-cell imaging is a highly used technique to study cell migration and dynamics over time. The image analysis workflow of volumetric (3D) imaging of cells via fluorescence microscopy starts with an accurate detection and segmentation of cells followed by cell tracking and track analysis. Broadly speaking the task of segmentation can be separated into semantic segmentation (classifying pixels as background or pixels belonging to the cell) or instance segmentation (classifying pixels belonging to individual cells by assigning a unique label to each cell). Segmentation is complicated due to presence of multiple objects in the image, overlapping object pixels and non-homogeneous intensity distribution. Several methods have been proposed for such automated detection and segmentation tasks such as the traditional intensity based thresholding, watershed transform [BM18] and of recent machine learning methods based on random-forest classifiers and support vector machines [BKK+19]. It was shown in [RHH20] that conventional computer vision and machine learning based techniques alone will almost always lead to sub-optimal segmentation and that methods based on deep learning have improved the accuracy of segmentation for natural and biomedical images alike. For the purpose of semantic segmentation U-Net [RFB15] has emerged as the most widely used network for biological applications. This network also forms the backbone of another successful network to do cell nuclei segmentation in 3D, Stardist [SWBM18] [WSH+20]. Stardist directly predicts a shape representation as star-convex polygons for cell nuclei in 2D and 3D. However, cell membrane segmentation is especially challenging as opposed to nuclei segmentation due to fewer boundary pixels and the need to separate touching cells. To predict cell contours together with cell centroids, Eschweiler et al. proposed a 3D U-Net network using centroids as seeds for watershed in 3D confocal microscopy images [ESC+18]. The drawback of this approach is misclassification due to sub-optimal seeding. Another approach proposed by Wolny et al., is to directly predict the cell boundaries using a 3D U-Net followed by a volume partitioning algorithm to segment each cell based on the boundary prediction [WCV+20]. This approach requires well defined cell boundaries, which may create segmentation errors in low signal-to-noise imaging conditions.

To address the issues with existing segmentation algorithms just described, we developed Vollseg. In brief we use Stardist in 3D to obtain a star convex shape approximation for the cells and extract the cell centroids from these polygons. We also train a 3D U-Net model to obtain a semantic segmentation map of the cells. We then perform a marker controlled watershed on the probability map of Stardist using the U-Net segmentation as a mask image to prevent the overflow of segmentation regions. To avoid the error of sub-optimal seeding we developed a seed pooling approach taking advantage of strength of both the Stardist and U-Net networks. We benchmark our segmentation result on a challenging dataset comprised of epithelial cells of mouse embryonic mammary glands with membrane labelling. These cells are highly irregular in shape and have a low signal-to-noise ratio to obtain an accurate segmentation only based on the boundary information. Using this dataset, we obtain different metrics showing that our approach is able to obtain shape approximation for the overlapping cells that go beyond the star convex shape. The complete segmentation pipeline is illustrated in Figure 1.

For analysis of the cell migration behavior we need to reliably track the cells and obtain certain attributes such as signal intensity or changes over time of the distance between the cells and tissue boundary. Cell tracking is challenging due to erratic volumetric motion, occlusion and cell divisions. Tracking using only the

* Corresponding author: varun.kapoor@curie.fr
‡ Institut Curie, PSL Research University, Sorbonne University, CNRSUMR 3215, INSERM U934, Genetics and Developemental Biology, Paris, France.

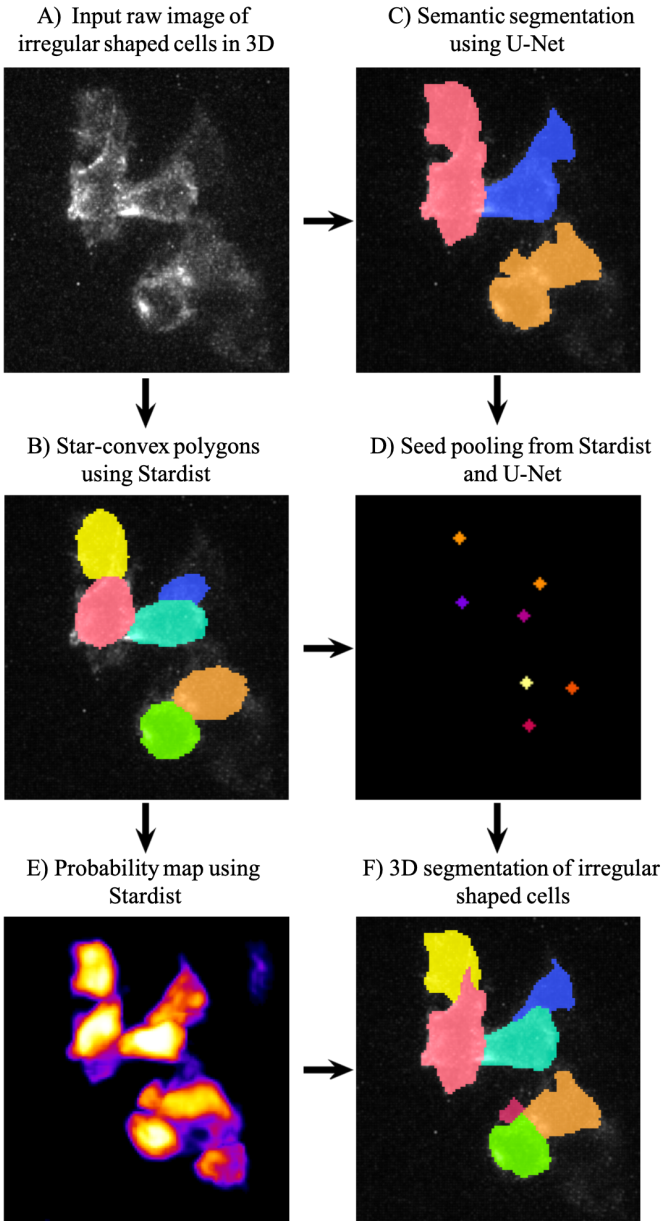

Fig. 1: *Schematic representation showing the segmentation approach used in VollSeg. First, we input the raw fluorescent image in 3D (A) and pre-process it to remove noise. Next, we obtain the star convex approximation to the cells using Stardist (B) and the U-Net prediction labelled via connected components (C). We then obtain seeds from the centroids of labelled image in B, for each labelled region of C in order to create bounding boxes and centroids. If there is no seed from B in the bounding box region from U-Net, we add the new centroid (shown in yellow) to the seed pool (D). Finally, we do a marker controlled watershed in 3D using skimage implementation on the probability map shown in (E) to obtain the final cell segmentation result (F). All images are displayed in Napari viewer with 3D display view.*

centroid information may lead to wrong cell assigmements, hence we need to include other cell attributes such as the shape and intensity information while making the links between the cells in successive time frames. Trackmate is a popular tracking software that uses customizable cost matrix for solving the linear assignment problem and uses Jaqman linker as a second step to link segments of dividing and merging cells [TPS+17]. In this paper, we introduce BTrackmate, a Fiji/ImageJ plugin to track the previously segmented cells. The major advantage of BTrackmate is the ability to track the cells inside a tissue. It allows the input of the cell and tissue segmentation image files and/or a csv file of the cell attributes. Furthermore, we also add some biological context in the tracking process where after segment linking is done a track inspector removes segments that are shorter than a user defined time length. Such short segments are unlikely to be true division events if they are too short and manually removing them can be tedious when many tracks are present. The users can choose this parameter in time units and can set it to 0 if removing such short segments is not required.

Finally, the tracking results obtained with BTrackmate are saved as an xml file that can be re-opened in an Euler angle based viewer in python called Napari, allowing volumetric viewing of the tracked cells using the track layer feature [UVCL20]. We made a python package called napatrackmater to export the track xml file as tracks layer in Napari for dividing and non-dividing tracks. We provide a customized Napari widget to view selected tracks and obtain their cell migration attributes.

Material and Methods

Preparation of the dataset

We used fluorescent microscopy images of mouse embryonic mammary glands stabilized in an ex vivo culture previously collected in the laboratory of Dr. S. Fre at Institut Curie. All images were acquired with an inverted confocal laser scanning microscope (e.g. Zeiss LSM780/880) equipped with long-working distance objectives to acquire high-resolution 3D image stacks. We acquired images of pixel size (22, 512, 512) with calibration of (3, 0.52, 0.52) micrometer. The quality at which these images are acquired is determined by the spatial resolution of the used optical device, desired temporal resolution, duration of the experiment and depth of the acquired Z-stacks. We perform unsupervised image denoising [KBJ19] on our dataset, an algorithm we chose based on its performance compared to other methods [Ric72], [Luc74]. Post-restoration of the 3D images, we developed a method to perform the segmentation of the cells using deep learning techniques. We created a training dataset with hand drawn segmentation of 14 Z-stacks. We performed data augmentation on the microscopy images by denoising, adding Poisson and Gaussian noise, random rotations and flips to create 700 Z-stacks. We chose a patch size of (16, 128, 128) and created 11,264 patches for training Stardist and U-Net network. For the Stardist network we chose 192 rays to have a better shape resolution for the irregular shaped cells.

Parameter Setting

Stardist predicts object instances based on probability threshold and non maximal suppression threshold to merge overlapping predictions. These parameters can be automatically determined using the optimize threshold program that we provide with the segmentation package. Higher values of the probability threshold yield fewer object instances, but avoids false positives. Higher

values of the overlap threshold would lead to oversegmentation. We used 32 Z-stacks to determine the optimal parameters of probability threshold of 0.76 and non maximal suppression threshold of 0.3.

Segmentation

As illustrated in Figure 1, we first obtain the centroids of the star convex approximated cell shapes and create a seed pool with these centroid locations. Even with the optimized threshold values we find that the seeds can be sub-optimal as many cells instances with low signal are missed. In order to make the seed pool optimal we use the U-Net prediction to obtain a binary image of semantic segmentation, perform connected component analysis to label the image and obtain bounding boxes (computed using scikit-image [vdWSN+14], version 0.18.x) for each label in 3D. For each bounding box we search for a seed from the Stardist predicted seed pool. If a Stardist seed is found inside the bounding box, the centroid of the U-Net predicted bounding box is rejected else the centroid is added to the seed pool to make a complete set of seeds that we use to start a watershed process in 3D. We use the probability map of Stardist to start the watershed process to obtain a better shape approximation for the irregular shaped cells that goes beyond the star convex shape.

The code for the merging of U-Net and Stardist seeds is the following:

```
def iou3D(box_unet, centroid_star):

  ndim = len(centroid_star)
  inside = False

  Condition = [Conditioncheck(centroid_star, box_unet,
   p, ndim)
   for p in range(0,ndim)]

  inside = all(Condition)

  return inside

def Conditioncheck(centroid_centroid, box_unet,
       p, ndim):

  condition = False

  if centroid_star[p] >= box_unet[p]
  and centroid_star[p] <= box_unet[p + ndim]:

       condition = True

  return condition
```

The code for doing watershed in 3D using the complete set of seeds on the probability map of Stardist is the following:

```
def WatershedwithMask3D(Image, Label, mask, grid):
  #Image = ProbabilityMap of Stardist
  #Label = Label segmentation image of Stardist
  #Mask = U-Net predicted image post binarization
  properties = measure.regionprops(Label, Image)
  binaryproperties =
  measure.regionprops(label(mask), Image)
  Coordinates = [prop.centroid for prop in properties]
  BinaryCoordinates = [prop.centroid for
  prop in binaryproperties]
  Binarybbox =
  [prop.bbox for prop in binaryproperties]
  Coordinates = sorted(Coordinates ,
  key=lambda k: [k[0], k[1], k[2]])

  if len(Binarybbox) > 0:
       for i in range(0, len(Binarybbox)):
            box = Binarybbox[i]
            inside = [iou3D(box, star)
            for star in Coordinates]

            if not any(inside) :
                 Coordinates.append(BinaryCoordinates[i])


  Coordinates.append((0,0,0))
  Coordinates = np.asarray(Coordinates)
  coordinates_int = np.round(Coordinates).astype(int)

  markers_raw = np.zeros_like(Image)
  markers_raw[tuple(coordinates_int.T)] = 1
  + np.arange(len(Coordinates))
  markers = morphology.dilation(
  markers_raw.astype('uint16'), morphology.ball(2))

  watershedImage = watershed(-Image, markers,
  mask = mask.copy())
  return watershedImage, markers
```

Performance Metrics

Accuracy of segmentation results is assesed by comparing the obtained labels to the ground truth (GT) labels. The most commonly used metric is to compute intersection over union (IOU) score between the predicted and the GT label image. We define GT, labels and IOU score as:

$GT = \{gt\}$, $SEG = \{seg\}$ are two sets of segmented objects.

$IOU(a,b)$ is the value of the IOU operation between two segmented objects a and b.

A threshold score value $\tau \in [0,1]$ is used to determine the true positive (TP), false positives (FP) and false negatives (FN) defined as:

$$TP = \{seg \in SEG, \exists\ gt \in GT,\ IOU(gt,seg) > \tau\}$$

$$FP = \{seg \in SEG, \forall\ gt \in GT,\ IOU(gt,set) < \tau\}$$

$$FN = \{gt \in GT, \forall\ seg \in SEG,\ IOU(gt,seg) < \tau\}$$

We use the Stardist implementation to compute accuracy scores which uses the hungarian method (scipy implementation) [Kuh55] to compute an optimal matching to do a one to one assingement of predicted label to GT labels. This implementation avoids finding multiple TP for a given instance of GT. We also compute precision (TP/(TP + FP)), recall (TP / (TP + FN)), F1 score (geometric mean of precision and recall) and accuracy score $AP_\tau = \frac{TP_\tau}{TP_\tau + FP_\tau + FN_\tau}$. To evaluate the accuracy of our method in resolving the shape of the cells we compute the mean squared error (MSE) and structural similarity index measurment (SSIM) between the GT and obtained segmentation images post-binarization operation on the obtained instance segmentation maps. MSE shows a low score if the image is structurally closer to GT. SSIM score is higher if the two images are structurally more similar to each other.

Detailed Procedure

The software package we provide comes with training and prediction notebooks for training the base U-Net and Stardist networks on your own dataset. We provide jupyter notebooks to do so on local GPU servers and also on Google Colab.

Network Training: In the first Jupyter notebook we create the dataset for U-Net and Stardist training. In the first cell of the notebook the required parameters are the path to your data that contains the folder of Raw and Segmentation images to create training pairs. Also to be specified is the name of the generated npz file along with the model directory to store the h5 files of the trained model and the model name.

```
Data_dir = '/data/'
NPZ_filename = 'VollSeg'
Model_dir = '/data/'
Model_Name = 'VollSeg'
```

The model parameters are specified in the next notebook cell. These parameters are described as follows:

1) NetworkDepth = Depth of the network, with each increasing depth the image is downsampled by 2 hence the XYZ dimension of the data / 2^depth has to be greater than 1.
2) Epochs: training for longer epochs ensures a well converged network and requires longer GPU runtimes.
3) Learning rate is the parameter which controls the step size used in the optimization process and it should not be greater than 0.001 at the start of the training.
4) Batch size controls the number of images used for doing stochastic gradient descent and is a parameter limited by the GPU memory available, batch size < 10 should be optimal.
5) Patch X, Y, Z is the size used for making patches out of the image data. The original image is broken down into patches for training. Patch size is chosen based on having enough context for the network to learn the details at different scales.
6) Kernel is the receptive field of the neural network, usual choices are 3, 5 or 7. This is the size of the convolutional kernel used in the network.
7) n_patches_per_image is the number of patches sampled for each image to create the npz file, choose an optimal value so that the file fits in the RAM memory.
8) Rays stand for the number of rays used to learn the distance map, low rays decreases the spatial resolution and high rays are able to resolve the shape better.
9) use_gpu_opencl is a boolean parameter that is set true if you want to do some opencl computations on the GPU, this requires GPU tools python package.
10) Before starting the U-Net training an npz file containing the paried Raw and Binary segmentation images needs to be created, by setting GenerateNPZ = True such a file is created.
11) If there are multiple GPU's available, the training of U-Net and Stardist can be split between the GPU's. Set TrainUNET = True for training a U-Net network, create a copy of the notebook and only set TrainSTAR = True for training a Stardist network. If there are no multiple GPU's available, set all of these parameters in 10) and 11) to be True to create and train both the networks in a single notebook run.

The code to set the parameters is the following:

```
#Network training parameters
NetworkDepth = 3
Epochs = 100
LearningRate = 1.0E-4
batch_size = 5
PatchX = 128
PatchY = 128
PatchZ = 16
Kernel = 3
n_patches_per_image = 16
Rays = 192
startfilter = 48
use_gpu_opencl = True
GenerateNPZ = True
TrainUNET = False
TrainSTAR = False
```

After the network has been trained it will save the configuration files of the training for both the networks along with the weight vector file as h5 files that will be used by the prediction notebook. For running the network prediction on XYZ shape images use the prediction notebook either locally or on Colab. In this notebook you only have to specify the path to the image and the model directory. The only two parameters to be set here are the number of tiles (for creating image patches to fit in the GPU memory) and min_size in pixel units to discard segmented objects below that size. We perform the watershed operation on the probability map as a default. However, this operation can also be changed to use the distance map coming out of Stardist prediction instead by setting 'UseProbability' variable to false. The code below operates on a directory of XYZ shape images:

```
ImageDir = 'data/tiffiles/'
Model_Dir = 'data/'
SaveDir = ImageDir + 'Results/'
UNETModelName = 'UNETVollSeg'
StarModelName = 'VollSeg'
NoiseModelName = 'NoiseVoid'

UnetModel = CARE(config = None,
name = UNETModelName,
basedir = Model_Dir)
StarModel = StarDist3D(config = None,
name = StarModelName,
basedir = Model_Dir)
NoiseModel = N2V(config=None,
name=NoiseModelName,
basedir=Model_Dir)

Raw_path =
os.path.join(ImageDir, '*.tif')
filesRaw =
glob.glob(Raw_path)
filesRaw.sort
min_size = 50
n_tiles = (1,1,1)
for fname in filesRaw:

     SmartSeedPrediction3D(ImageDir,
     SaveDir, fname,
     UnetModel, StarModel, NoiseModel,
     min_size = min_size,
     n_tiles = n_tiles,
     UseProbability = False)
```

Tracking

After we obtain the segmentation using VollSeg, we create a csv file of the cell attributes that include their location, size and volume inside a region of interest. For large datasets memory usage could be of concern while loading the images into memory, hence inputs via csv could prove helpful. Tracking is performed in ImageJ/Fiji, an image processing package. We developed our code over the existing tracking solution called Trackmate [TPS+17]. Trackmate uses linear assignment problem (LAP) algorithm to do linking of the cells and uses Jaqman linker for linking the segments for dividing and merging trajectories. It also provides other trackers such as the Kalman filter to do tracking of non-dividing cells.

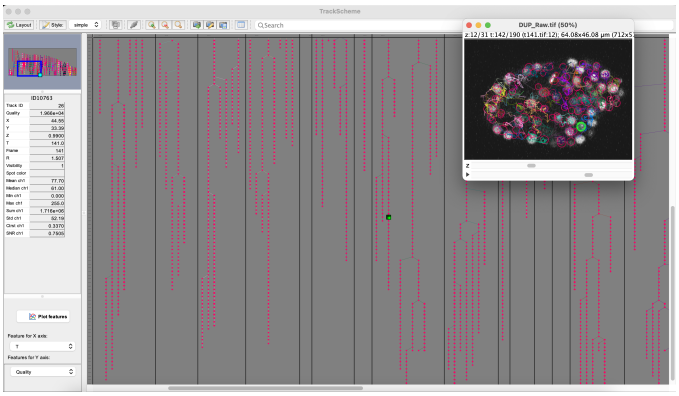

Fig. 2: *Trackscheme display for the C. elegans dataset.*

Trackmate comes with a fully interactive track editing interface with graph listener to show the selected cell in the trackscheme and vice versa, to click on the graph and have the selected cell being highlighted in the image, making the process of track editing interactive. Post-editing the tracks are saved as an xml file which can then be loaded back into the program to do more track editing if needed. When a cell divides, the track is splitted up in two tracklets. In order to aid in track editing, we introduced a new parameter of minimum tracklet length to remove tracklets in a track that are short in the time dimension. This introduces a biological context of not having very short trajectories, reducing the track editing effort to correct for the linking mistakes made by the program. For testing our tracking program we used a freely available dataset from the cell tracking challenge of a developing C. elegans embryo [Cel] [MBB+08]. Using our software we can remove cells from tracking which do not fit certain criteria such as being too small (hence most likely a segmentation mistake) or being low in intensity or outside the region of interest such as when we want to track cells only inside a tissue. For this dataset we kept 12,000 cells and after filtering short tracks kept about 50 tracks with and without division events.

For this dataset the track scheme along with overlayed tracks is shown in Figure 2. Selected node in the trackscheme is highlighted in green and vice versa. Extensive manual for using the track editing is available on ImageJ/Fiji wiki [Tin].

Results

Quantitative Comparisons between Segmentation Methods

We compare our proposed VollSeg segmentation approach to two commonly used methods for cell segmentation of fluorescent microscopy images, 3D Stardist [SWBM18] [WSH+20] and 3D U-Net [RFB15]. A 3D cell rendering using all analyzed segmentation methods is shown in the Figure 3. Stardist in 3D was previously compared to other classical method, the IFT watershed, and it was shown to perform better than the classical method, hence we use Stardist as a baseline for comparison. To assess the performance of our segmentation, we compute the metrics described in material and methods section. VollSeg and Stardist methods perform at comparable accuracy, but higher than U-Net, as shown in Figure 4 A. This is expected, as U-Net can not perform instance segmentation of overlapping cells. In addition, when quantifying the F1-score in Figure 4 B, U-Net obtains the lowest score because it detects less TP segmented pixels in comparision to VollSeg and Stardist as shown in Figure 4 C. However, Stardist has the highest

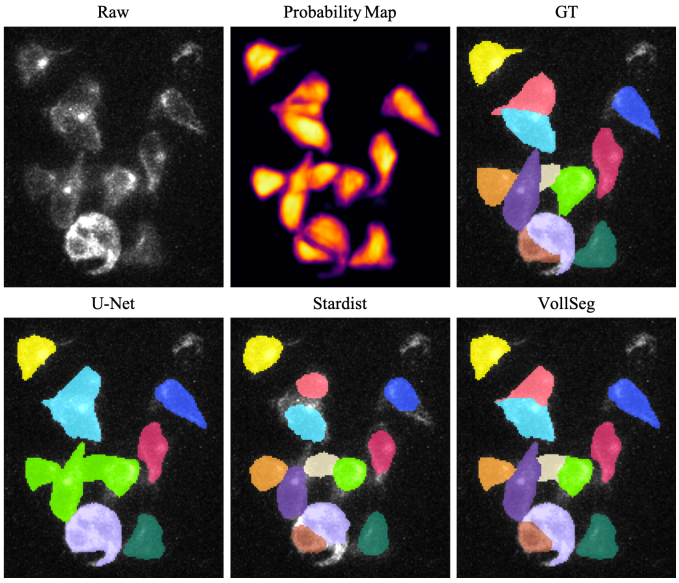

Fig. 3: *Visual 3D segmentation comparison between the Ground truth (GT) image, Stardist, U-Net and VollSeg results. The images are displayed in Napari viewer with 3D display view.*

mean squared error as it is unable to detect the irregular shape while U-Net and Vollseg have similar performance, as shown in Figure 5 A. This result can also be seen from structural similarity index measurement, shown in Figure 5 B. In conclusion, VollSeg is able to strength the shape accuracy from U-Net and the ability to separate the overlapping instances from Stardist.

Track Analysis

After obtaining the tracks from BTrackmate, we save them as Trackmate xml file, which contains the information about all the cells in a track. Since the cells can be highly erratic in their volumetric motions, we use Napari, an Euler angle based viewer, to visualize such tracks from different reference positions. We made a python package to export the xml files previously saved in ImageJ/Fiji and convert them into the tracks layer of Napari. We made a customised widget based graphic user interface (GUI) to view selected tracks, display the track information and save the cell track along user selected view, as shown in Figure 6 A. On the top left panel, the image and tracks layer properties are displayed and can be changed (1). In the bottom left, there is a dropdown menu enlisting all the tracks (2). Users can select the track to be displayed in the central window and it can be switched between the hyperstack and the 3D view (3). The user can also choose to view all the tracks at once and then toggle the visibilty of the tracks using the eye icon next to the image and tracks layer (4). On the top right panel, we show two plots displaying the track information (5). The 3D central view can be rotated and translated to view the tracks along the plane of motion of the cells and the selected view can be saved as an animation using the bottom right animation panel (6). For the cells that divide we show the intensity variation and associated fast fourier transform for each tracklet.

We provide two example jupyter notebooks with the package. In the first one we compute the cell distance from the tissue boundary change over time for dividing and non-dividing trajectories. The user selects a track of interest and it displays two plots next to the track view that show the distance change over time for the whole track (non-dividing trajectory) and the starting and end

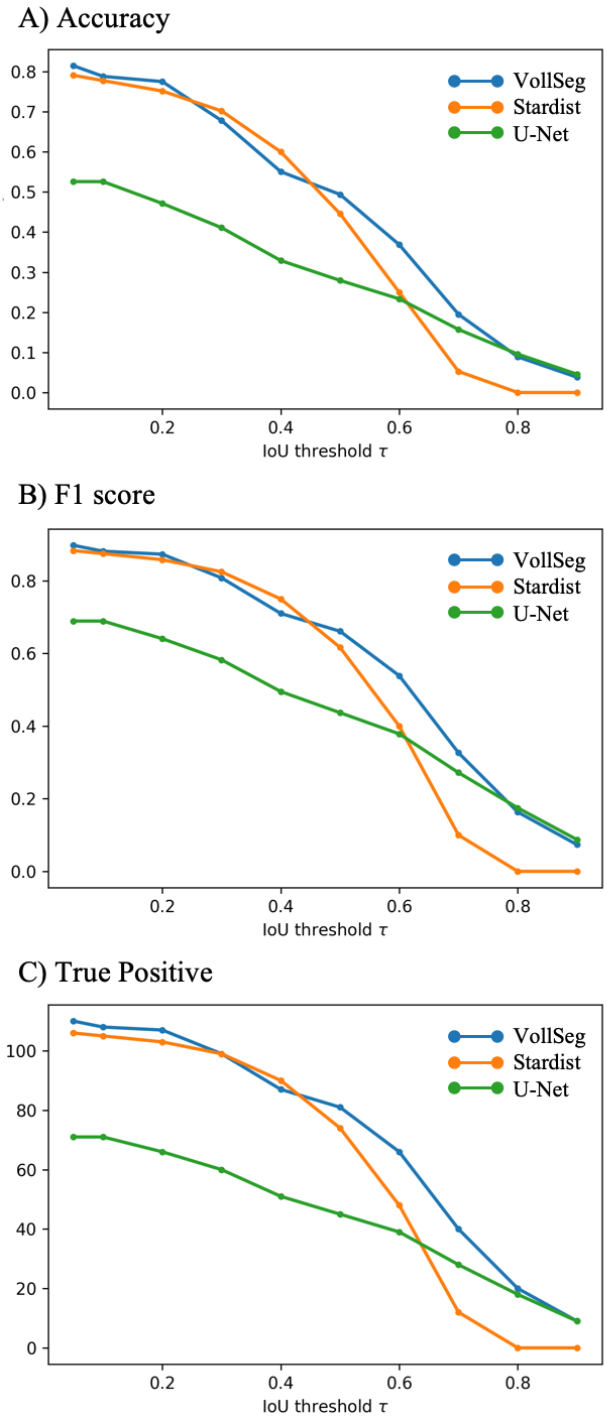

Fig. 4: *Segmentation comparision metrics between VollSeg (in blue), Stardist (in orange) and U-Net (in green). We plot (A) accuracy (as percentage), (B) F1 score (as percentage) and (C) true positive rates (as number of pixels) for all the networks.*

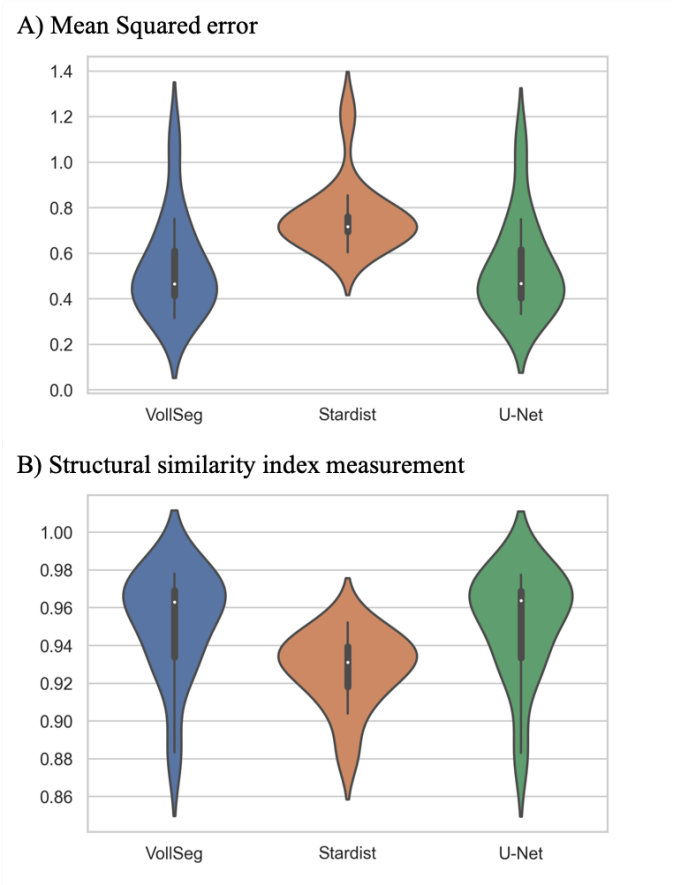

Fig. 5: *We plot Mean Squared error (MSE) (A) and Structural similarity index measurement (SSIM) (B) comparing between VollSeg (in blue), Stardist (in orange) and U-Net (in green).*

location of the cells, as shown in Figure 7. For the tracks with multiple events of cell division we show the distance change over time of each tracklet. In the localization plot the parent tracklet start and end location is shown in green while all the daughter cells start and end locations are shown in red. In the second example notebook, the plots show intensity change in the track over time along with the associated frequency of intensity oscillation present in each tracklet. The frequency associated with each tracklet is computed using the scipy implementation of fast fourier transform. The results of track analysis can be saved as plots, mp4 files of the track animation or csv files.

Conclusions

We have presented a workflow to do segmentation, tracking and track analysis of cells in 3D with irregular shape and intensity distribution. For performing segmentation we developed VollSeg, a jupyter notebook based python package that combines the strengths of semantic and instance deep learning segmentation methods. Post-segmentation we create a csv file containing the information about the cells inside a region of interest which serves as an input to Btrackmate, the ImageJ/Fiji plugin we created for doing the tracking. The tracking software uses existing track editing interface of Trackmate and saves the track information as an xml file. To view and analyze such volumetric tracks we created napatrackmater, a python package to export such trajectories as

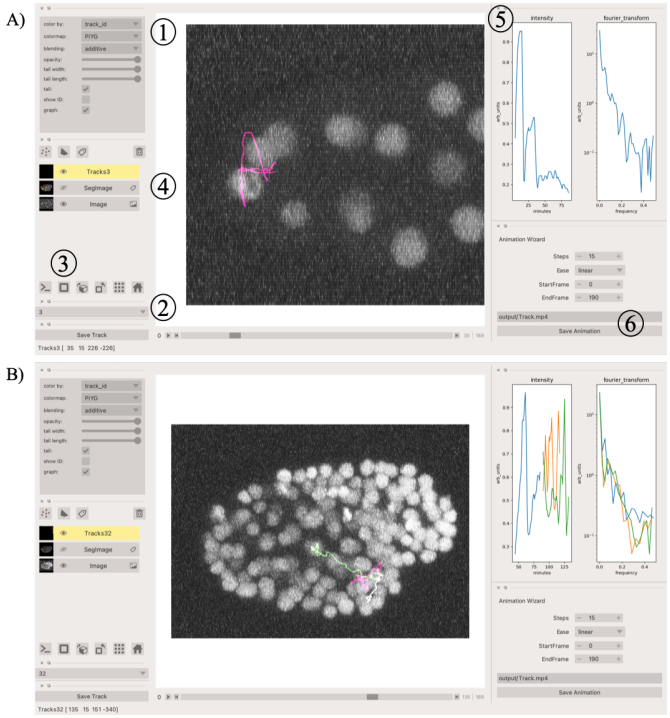

Fig. 6: *Napari widget to view tracks and plot track information in non-dividing trajectories (A) and dividing trajecrtories (B). For the selected track we see the intensity change over time and its associated fast Fourier transform.*

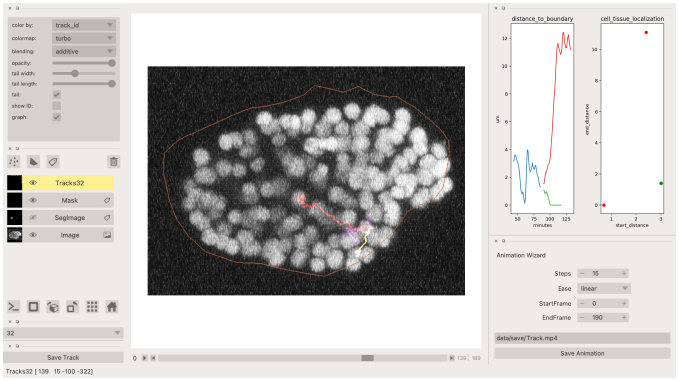

Fig. 7: *Napari widget to analyze the distance of the cell to the boundary. The left plot displays the distance of the daughter cells to the boundary, while the right plot shows the start and end distance localization of the mother cell (in green) and daughter cells (in red).*

track layer of Napari and we provide jupyter notebook based enviornment for track analysis with two example notebooks.

The tools that we present here can also be useful for segmentation of cells coming from other organisms or imaging modalities (transmitted light and light sheet imaging) as our method can be applied to segment cells that go beyond the star convex polyhedra.

Acknowledgements

We acknowledge the Cell and Tissue Imaging Platform (PICT-IBiSA) of the Genetics and Developmental Biology Department (UMR3215/U934) at Institut Curie, member of the French National Reserch infrastructure France-Bioimaging (ANR-10-INBS-04). We thank specially Olivier Renaud for supporting the software development. We are grateful to Dr Silvia Fre for support and constructive discussions. We thank Leo Guginard for insightful comments about the manuscript. V.K is supported by Labex DEEP at Institut Curie (ANR-11- LBX0044 grant). C.C is supported by funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 666003.

Author Contributions

V.K wrote the code; C.C performed the image acquisition of the used dataset and created labelled training dataset in 3D; V.K and C.C wrote the manuscript.

References

[BKK+19] Stuart Berg, Dominik Kutra, Thorben Kroeger, Christoph N. Straehle, Bernhard X. Kausler, Carsten Haubold, Martin Schiegg, Janez Ales, Thorsten Beier, Markus Rudy, Kemal Eren, Jaime I. Cervantes, Buote Xu, Fynn Beuttenmueller, Adrian Wolny, Chong Zhang, Ullrich Koethe, Fred A. Hamprecht, and Anna Kreshuk. ilastik: interactive machine learning for (bio)image analysis. *Nature Methods*, September 2019. doi:10.1038/s41592-019-0582-9.

[BM18] S. Beucher and F. Meyer. The morphological approach to segmentation: The watershed transformation. 2018. doi:10.1201/9781482277234-12.

[Cel] Waterston lab, university of washington, seattle, wa, usa. http://celltrackingchallenge.net/3d-datasets/.

[ESC+18] Dennis Eschweiler, Thiago V. Spina, Rohan C. Choudhury, Elliot Meyerowitz, Alexandre Cunha, and Johannes Stegmaier. Cnn-based preprocessing to optimize watershed-based cell segmentation in 3d confocal microscopy images, 2018. arXiv:1810.06933, doi:10.1109/isbi.2019.8759242.

[Git] https://github.com/kapoorlab/vollseg, https://github.com/kapoorlab/napatrackmater, https://github.com/kapoorlab/btrackmate.

[KBJ19] Alexander Krull, Tim-Oliver Buchholz, and Florian Jug. Noise2void-learning denoising from single noisy images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2129–2137, 2019. doi:10.1109/cvpr.2019.00223.

[Kuh55] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109, doi:10.1002/nav.3800020109.

[Luc74] L.B. Lucy. An iterative technique for the rectification of observed distributions. *The Astronomical Journal 7*, 79:745, 1974. doi:10.1086/111605.

[MBB+08] John Isaac Murray, Zhirong Bao, Thomas J Boyle, Max E Boeck, Barbara L Mericle, Thomas J Nicholas, Zhongying Zhao, Matthew J Sandel, and Robert H Waterston. Automated analysis of embryonic gene expression with cellular resolution in c. elegans. *Nature Methods*, 5(8):703–709, 2008. doi:10.1038/nmeth.1228.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[RHH20] Tobias M. Rasse, Réka Hollandi, and Peter Horvath. Opsef: Open source python framework for collaborative instance segmentation of bioimages. *Frontiers in Bioengineering and Biotechnology*, 8:1171, 2020. doi:10.3389/fbioe.2020.558880.

[Ric72] William Hadley Richardson. Bayesian-based iterative method of image restoration*. *J. Opt. Soc. Am.*, 62(1):55–59, Jan 1972. doi:10.1364/JOSA.62.000055.

[SWBM18] Uwe Schmidt, Martin Weigert, Coleman Broaddus, and Gene Myers. Cell detection with star-convex polygons. In *Medical Image Computing and Computer Assisted Intervention - MICCAI 2018 - 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II*, pages 265–273, 2018. doi:10.1007/978-3-030-00934-2_30.

[Tin] Jean-Yves Tinevez. Trackmate manual. https://imagej.net/TrackMate.

[TPS+17] Jean-Yves Tinevez, Nick Perry, Johannes Schindelin, Genevieve M. Hoopes, Gregory D. Reynolds, Emmanuel Laplantine, Sebastian Y. Bednarek, Spencer L. Shorte, and Kevin W. Eliceiri. Trackmate: An open and extensible platform for single-particle tracking. *Methods*, 115:80–90, 2017. Image Processing for Biologists. doi:10.1016/j.ymeth.2016.09.016.

[UVCL20] Kristina Ulicna, Giulia Vallardi, Guillaume Charras, and Alan R. Lowe. Automated deep lineage tree analysis using a bayesian single cell tracking approach. *bioRxiv*, 2020. arXiv:https://www.biorxiv.org/content/early/2020/09/10/2020.09.10.276980.full.pdf, doi:10.1101/2020.09.10.276980.

[vdWSN+14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014. doi:10.7717/peerj.453.

[WCV+20] Adrian Wolny, Lorenzo Cerrone, Athul Vijayan, Rachele Tofanelli, Amaya Vilches Barro, Marion Louveaux, Christian Wenzl, Susanne Steigleder, Constantin Pape, Alberto Bailoni, Salva Duran-Nebreda, George Bassel, Jan U. Lohmann, Fred A. Hamprecht, Kay Schneitz, Alexis Maizel, and Anna Kreshuk. Accurate and versatile 3d segmentation of plant tissues at cellular resolution. *bioRxiv*, 2020. arXiv:https://www.biorxiv.org/content/early/2020/01/18/2020.01.17.910562.full.pdf, doi:10.1101/2020.01.17.910562.

[WSH+20] Martin Weigert, Uwe Schmidt, Robert Haase, Ko Sugawara, and Gene Myers. Star-convex polyhedra for 3d object detection and segmentation in microscopy. In *The IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2020. doi:10.1109/WACV45572.2020.9093435.