



#SciPy2018
Austin, Texas July 9-15

Scientific Computing with Python

Proceedings of the 17th Python in Science Conference

July 9 - July 15 • Austin, Texas

Fatih Akici
David Lippa
Dillon Niederhut
M Pacer

PROCEEDINGS OF THE 17TH PYTHON IN SCIENCE CONFERENCE

Edited by Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer.

SciPy 2018
Austin, Texas
July 9 - July 15, 2018

Copyright © 2018. The articles in the Proceedings of the Python in Science Conference are copyrighted and owned by their original authors

This is an open-access publication and is distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

For more information, please see: <http://creativecommons.org/licenses/by/3.0/>

ISSN:2575-9752
<https://doi.org/10.25080/Majora-4af1f417-018>

ORGANIZATION

Conference Chairs

PRABHU RAMACHANDRAN, Enthought Inc. & IIT Bombay
SERGE REY, Arizona State University

Program Chairs

LORENA BARBA, George Washington University
GIL FORSYTH, Capital One

Communications

PAUL IVANOV, Bloomberg

Birds of a Feather

NELLE VAROQUAUX, Berkeley Institute for Data Science
JESSICA HAMRICK, Deep Mind

Proceedings

FATIH AKICI, ACE Cash Express
DAVID LIPPA, Amazon
DILLON NIEDERHUT, Enthought
M PACER, Netflix

Financial Aid

CELIA CINTAS, CONICET
SCOTT COLLIS, Argonne National Laboratory
PAT KELLY, Los Alamos National Laboratory
ERIC MA, MIT

Tutorials

ALEXANDRE CHABOT-LECLERC, Enthought
MIKE HEARNE, USGS
BEN ROOT, Atmospheric and Environmental Research, Inc.

Sprints

RYAN MAY, University Corporation for Atmospheric Research
JONATHAN ROCHER, KBI Biopharma
CHARLYE TRAN, Lewis University and The Recurse Center

Diversity

JACKIE KAZIL, Capital One
JULIE KRUGLER HOLLEK, Twitter

Activities

KYLE NIEMEYER, Oregon State University
JULIA PASQUARELLA, Enthought

Sponsors

JILL COWAN, Enthought

Financial

BILL COWAN, Enthought
JODI HAVRANEK, Enthought

Logistics

JILL COWAN, Enthought

Proceedings Reviewers

ALBERTO ANTONIETTI
ALEJANDRO WEINSTEIN
ALEXANDRA ABATE
ANGELOS KRYPTOS
ANKUR ANKAN
ANNETTE GREINER
AVIPSA ROY
BILLY OKAL
BRIAN MCFEE
CHANDAN BOSE
CHITARANJAN MAHAPATRA
CHRIS CALLOWAY
CYRUS HARRISON
DANA FARBER
DAVID LIPPA
DEMBA BA
DILLON NIEDERHUT
FATIH AKICI
GUY TEL-ZUR
HOMIN LEE
JAIME ARIAS ALMEIDA
JAMES BEDNAR
JASON GROUT
JEREMIAH JOHNSON
JOHN LEEMAN
KRISHNA NEUPANE
KYLE KELLEY
MARIANNE HOOGEVEEN
MARK FENNER
MATT ROCKLIN
MEGAN SOSEY
MIKE SARAHAN
NICHOLAS MALAYA
NICOLÁS GUARÍN-ZAPATA
PATRICK HUCK
PAULINE BARMBY
RAVI SIVALINGAM
RICARDO BARROS LOURENÇO
RICARDO FERRAZ LEAL
SCOTT SIEVERT
SEETHA KRISHNAN
STEFAN VAN DER WALT
SUZANNE STATHATOS
TOM AUGSPURGER
TUMMALAPALLI SUDHAMSH REDDY
TZU-CHI YEN
YINGWEI YU

SCHOLARSHIP RECIPIENTS

JAMES BOURBEAU, University of Wisconsin, Madison
JAMES ALEXANDER BRANHAM, University of Texas at Austin
ROBERTO COLISTETE JUNIOR, Universidade Federal do Espirito Santo
FILIPE FERNANDES, conda-forge
KENNETH LYONS, University of California, Davis
UNIVERSITY OF OREGON, Zachary Sailer
MRIDUL SETH, BITS Pilani, Goa
SCOTT SIEVERT, University of Wisconsin, Madison

JUMP TRADING AND NUMFOCUS DIVERSITY SCHOLARSHIP RECIPIENTS

HANNAH AIZENMAN, Matplotlib
MARIANNE CORVELLEC, Institute for Globally Distributed Open Research and Education
KADAMBARI DEVARAJAN, University of Massachusetts at Amherst
JULIE LAVOIE,
YEE NG, University of Texas at Austin
PARUL SETHI, University of Delhi
MALVIKA SHARAN, EMBL, Heidelberg
CLARE SLOGGETT, Melbourne Bioinformatics, University of Melbourne
CHAYA STERN, Memorial Sloan Kettering Cancer Center
HORACIO ANDRES VARGAS GUZMAN, Max Planck Institute for Polymer Research

CONTENTS

Dynamic Social Network Modeling of Diffuse Subcellular Morphologies <i>Andrew Durden, Allyson T Loy, Barbara Reaves, Mojtaba Fazli, Abigail Courtney, Frederick D Quinn, S Chakra Chennubhotla, Shannon P Quinn</i>	1
Cloudknot: A Python Library to Run your Existing Code on AWS Batch <i>Adam Richie-Halford, Ariel Rokem</i>	8
Equity, Scalability, and Sustainability of Data Science Infrastructure <i>Anthony Suen, Laura Norén, Alan Liang, Andrea Tu</i>	15
Composable Multi-Threading and Multi-Processing for Numeric Libraries <i>Anton Malakhov, David Liu, Anton Gorshkov, Terry Wilmarth</i>	18
The Econ-ARK and HARK: Open Source Tools for Computational Economics <i>Christopher D. Carroll, Alexander M. Kaufman, Jacqueline L. Kazil, Nathan M. Palmer, Matthew N. White</i>	25
Developing a Start-to-Finish Pipeline for Accelerometer-Based Activity Recognition Using Long Short-Term Memory Recurrent Neural Networks <i>Christian McDaniel, Shannon Quinn</i>	31
Practical Applications of Astropy <i>David Shupe, Frank Masci, Russ Laher, Ben Rusholme, Lee Armus</i>	41
EarthSim: Flexible Environmental Simulation Workflows Entirely Within Jupyter Notebooks <i>Dharhas Pothina, Philipp J. F. Rudiger, James A Bednar, Scott Christensen, Kevin Winters, Kimberly Pevey, Christopher E. Ball, Gregory Brener</i>	48
Safe handling instructions for missing data <i>Dillon Niederhut</i>	56
Text and data mining scientific articles with alloplos <i>Elizabeth Seiver, M Pacer, Sebastian Bassi</i>	61
Sparse: A more modern sparse array library <i>Hameer Abbasi</i>	65
Bringing ipywidgets Support to plotly.py <i>Jon Mease</i>	69
WrightSim: Using PyCUDA to Simulate Multidimensional Spectra <i>Kyle F Sunden, Blaise J Thompson, John C Wright</i>	77
Exploring the Extended Kalman Filter for GPS Positioning Using Simulated User and Satellite Track Data <i>Mark Wickert, Chiranth Siddappa</i>	84
Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets <i>Mark Wickert</i>	91
Organic Molecules in Space: Insights from the NASA Ames Molecular Database in the era of the James Webb Space Telescope <i>Matthew J. Shannon, Christiaan Boersma</i>	99
Harnessing the Power of Scientific Python to Investigate Biogeochemistry and Metaproteomes of the Central Pacific Ocean	106
Binder 2.0 - Reproducible, interactive, sharable environments for science at scale <i>Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osherooff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan-Kelley, Carol Willing</i>	113

Spatio-temporal analysis of socioeconomic neighborhoods: The Open Source Longitudinal Neighborhood Analysis Package (OSLNAP)	121
<i>Sergio Rey, Elijah Knaap, Su Han, Levi Wolf, Wei Kang</i>	
Design and Implementation of pyPRISM: A Polymer Liquid-State Theory Framework	129
<i>Tyler B. Martin, Thomas E. Gartner III, Ronald L. Jones, Chad R. Snyder, Arthi Jayaraman</i>	
A Bayesian's journey to a better research workflow	137
<i>Konstantinos Vamvourellis, Marianne Corvellec</i>	
Scalable Feature Extraction with Aerial and Satellite Imagery	145
<i>Virginia Ng, Daniel Hofmann</i>	
signac: A Python framework for data and workflow management	152
<i>Vyas Ramasubramani, Carl S. Adorf, Paul M. Dodd, Bradley D. Dice, Sharon C. Glotzer</i>	
Yaksh: Facilitating Learning by Doing	160
<i>Prabhu Ramachandran, Prathamesh Salunke, Ankit Javalkar, Aditya Palaparthi, Mahesh Gudi, Hardik Ghaghada</i>	

Dynamic Social Network Modeling of Diffuse Subcellular Morphologies

Andrew Durden^{||}, Allyson T Loy[¶], Barbara Reaves[‡], Mojtaba Fazli^{||}, Abigail Courtney[¶], Frederick D Quinn[‡], S Chakra Chennubhotla[§], Shannon P Quinn^{||***}



Abstract—The use of fluorescence microscopy has catalyzed new insights into biological function, and spurred the development of quantitative models from rich biomedical image datasets. While image processing in some capacity is commonplace for extracting and modeling quantitative knowledge from biological systems at varying scales, general-purpose approaches for more advanced modeling are few. In particular, diffuse organellar morphologies, such as mitochondria or actin microtubules, have few if any established spatiotemporal modeling strategies, all but discarding critically important sources of signal from a biological system. Here, we discuss initial work into building spatiotemporal models of diffuse subcellular morphologies, using mitochondrial protein patterns of cervical epithelial (HeLa) cells. We leverage principles of graph theory and consider the diffuse mitochondrial patterns as a social network: a collection of vertices interconnected by weighted and directed edges, indicating spatial relationships. By studying the changing topology of the social networks over time, we gain a mechanistic understanding of the types of stresses imposed on the mitochondria by external stimuli, and can relate these effects in terms of graph theoretic quantities such as centrality, connectivity, and flow. We demonstrate how the mitochondrial pattern can be faithfully represented parametrically using a learned mixture of Gaussians, which is then perturbed to match the spatiotemporal evolution of the mitochondrial patterns over time. The learned Gaussian components can then be converted to graph Laplacians, formally defining a network, and the changes in the topology of the Laplacians can yield biologically-meaningful interpretations of the evolving morphology. We hope to leverage these preliminary results to implement a bioimaging toolbox, using existing open source packages in the scientific Python ecosystem (SciPy, NumPy, scikit-image, OpenCV), which builds dynamic social network models from time series fluorescence images of diffuse subcellular protein patterns. This will enable a direct quantitative comparison of network structure over time and between cells exposed to different conditions.

Index Terms—Biomedical Imaging, Graph Theory, Social Networks

Introduction

Given the recent rise of fluorescence microscopy, and the subsequent proliferation of biomedical imaging data, live cell imaging

^{||} Department of Computer Science, University of Georgia, Athens, GA 30602 USA

[¶] Department of Microbiology, University of Georgia, Athens, GA 30602 USA

[‡] Department of Infectious Diseases, University of Georgia, Athens, GA 30602 USA

[§] Department of Computational and Systems Biology, University of Pittsburgh, Pittsburgh, PA 15232 USA

* Corresponding author: spq@uga.edu

** Department of Cellular Biology, University of Georgia, Athens, GA 30602 USA

Copyright © 2018 Andrew Durden et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

has become much more accessible. However, the growth in quantification and modeling of biological and biomedical phenomena has been uneven; "solid" morphologies such as cells and nuclei are much easier to automatically segment, track, and quantify than diffuse patterns induced by mitochondria or actin. There is a need for methodologies and software capable of autonomous tracking, segmentation, and quantification of spatiotemporal changes in these structures.

Understanding the spatiotemporal evolution of subcellular organelles in response to external stimuli and modeling this behavior is critical to understanding the effects of the stimuli on the internal state and configuration of the cell. This can have downstream implications in the development of targeted therapies. Recently, spatial covariance has been used to quantify gene expression correlation in image like matrices representing sequenced RNA [STS17]. Other recent work demonstrates the benefits of measuring covariance between subcellular structures to observe how coherent portions of the cells respond in tandem to external stimuli [VCL⁺17]. While this work used hand-crafted pixel-level thresholds and manual labeling of pixels into organelle groupings, it nonetheless represents the spirit of our work: developing quantifiable, data-driven spatiotemporal models of subcellular structures.

Our work focuses on both spatial and temporal covariance to better model and understand the response of subcellular structures to stimuli. To do this, we draw on graph theory and cast the punctate subcellular morphologies as instances of a social network. A recent study of brain activity used networks to create a quantitative measure of correlated activity in functional MRI (fMRI) images which could then easily be clustered [DGD⁺16]. There are many advantages of using a social network model for representing diffuse structures. It captures not only the overall spatial morphology and distribution of the protein pattern, but also intrinsically captures relationships between different spatial components of the pattern. Finally, by permitting the network to evolve over time, the changing properties of the social network can be interpreted biologically to describe different observed phenomena: just as "traditional" social networks evolve through the addition and deletion of connections between individuals, so do such events describe precisely how the morphology, both locally in one part of the cell, and globally across multiple cells, changes in response to stimuli.

We have begun by modeling the subcellular patterns of mitochondria in cervical epithelial (HeLa) cells. Mitochondria are dynamic organelles, which undergo continual rounds of fission

and fusion. These fission and fusion events are important for maintaining proper function and overall mitochondrial health [ZLN13] [WL16]. Mitochondrial fission allows for the turnover of damaged and the protection of healthy organelles. Additionally, mitochondrial fusion leads to the mixing of internal contents, which is important for responding to environmental needs [ZLN13] [KPSBW08].

The dynamics between fission and fusion creates a spectrum of mitochondrial morphologies. Imbalances between fission and fusion events generate phenotypes associated with mitochondrial dysfunction [ZLN13]. An excess of fission or dearth of fusion events results in fragmented mitochondria; in this phenotype, the mitochondrial network is fractured, and individual mitochondria exist in small spheres. Conversely, an overabundance of fusion or a lack of fission events generate hyperfused mitochondria; in this phenotype, the mitochondrial network is overconnected, and composed of long interconnected tubules [CSCI+08]. Recently, several bacterial species have been shown to cause mitochondrial perturbations during infection [SBS+11][FCGQR15]. Such unique morphologies should be detectable at a quantitative level using social network modeling.

Through social network modeling, we hope to build a more rapid and efficient method for identifying changes in size, shape, and distribution of mitochondria as well as other diffuse organelles. In this work, we present a proof-of-concept pipeline which segments cells with fluorescent stains on the mitochondria for individual analysis. Once the cells are segmented, we use a Gaussian Mixture Model (GMM) to parameterize the spatial distribution of the mitochondrial protein patterns at evenly-spaced time intervals, and allow the GMM parameters to update smoothly from the previous time point to the next. Finally, we demonstrate how the learned parameters of the GMM can be used to construct social networks for representing the mitochondria. The complete pipeline can be seen in Fig. 1.

Data

We have constructed a library of live confocal imaging videos that display the full spectrum of mitochondrial morphologies in HeLa cells, from fragmented to hyperfused. To visualize the mitochondria, HeLa cells were stably transfected with DsRed2-Mito-7 (DsRed2-HeLa), which fluorescently labels mitochondria with red emission spectra (a gift from Michael Davidson, Addgene plasmid #55838). All of our videos were taken using a Nikon AIR Confocal. Cells were kept in an imaging chamber that maintained 37 degrees C and 5% CO₂ for the duration of imaging. The resonant scanning head was used to capture an image every ten seconds for the length of the video. The resulting time series videos have more than 20,000 frames per video. Each frame is of dimensions of 512x512 pixels (Fig. 2).

Wild type mitochondrial morphology was captured by imaging DsRed2-HeLa cells in typical growth medium (DMEM plus 10 % fetal bovine serum) (Fig. 2, center). To generate the fragmented phenotype, cells were exposed to the pore-forming toxin listeriolysin O (LLO) at a final concentration of 6 nM (Fig. 2, left). Mitochondrial hyperfusion was induced through the addition of mitochondria division inhibitor-1 (mdivi-1) at a final concentration of 50 μ M (Fig. 2, right). These subsets with different known qualitative phenotypes serve as bases upon which to condition our quantitative analyses.

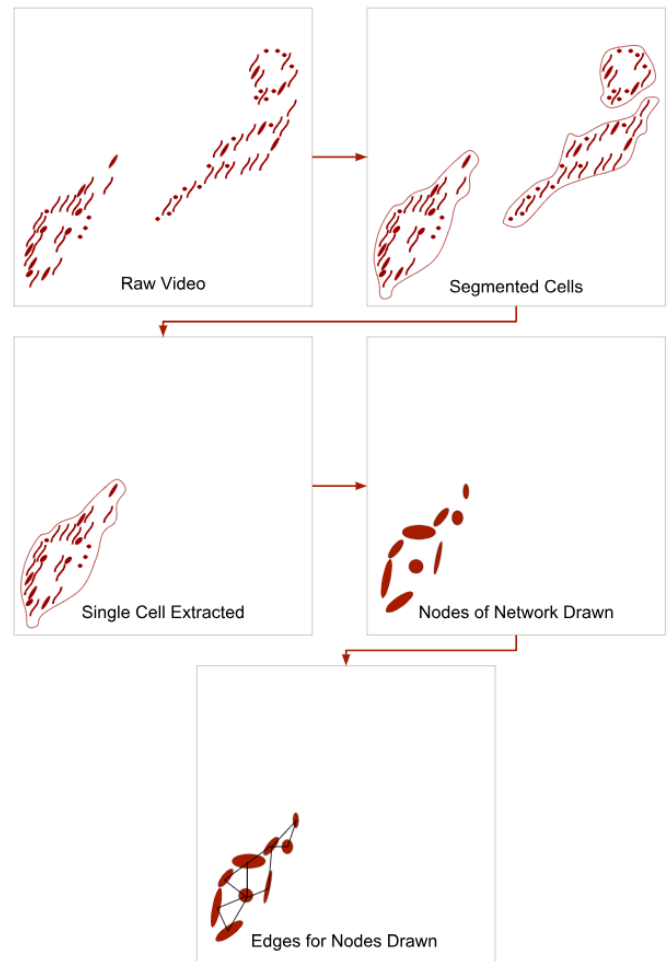


Fig. 1: An abstract representation of our proposed pipeline. The first frame represents the raw unsegmented image of mitochondria in three cells. The second frame demonstrates simultaneous segmentation, as a border is drawn around each cell. The third frame represents a single cell being extracted for analysis using the determined segmentation. The fourth frame shows a characteristic set of nodes determined by applying a mixture model to the distribution of fluorescent mitochondria. The final frame shows edges added to the nodes to complete the network structure. At this point in the pipeline, network analysis can be applied to the induced graph. These steps are applied to each frame of video allowing for fully temporal analysis. .

Segmentation Pipeline

In order to avoid systemic bias in our downstream analysis pipeline as a result of different videos containing a varied and unbounded number of cells, we chose to study each cell individually. This required segmenting each individual cell and studying its spatiotemporal dynamics in isolation from the others. While segmentation of cells from fluorescence or histology images is becoming very common, segmenting diffuse protein patterns--such as mitochondria--is much more challenging. We leveraged the fact that, given the small interval (10s) between frames of a video, overall movement between a given pair of subsequent frames would be minimal. We used deformable contours with slight updates from the previous frame to build out segmentation masks. However, the diffuse structure combined with the near overlap of cells in frames necessitated a "priming" of the segmentation pipeline with a hand-drawn mask at time 0. We used

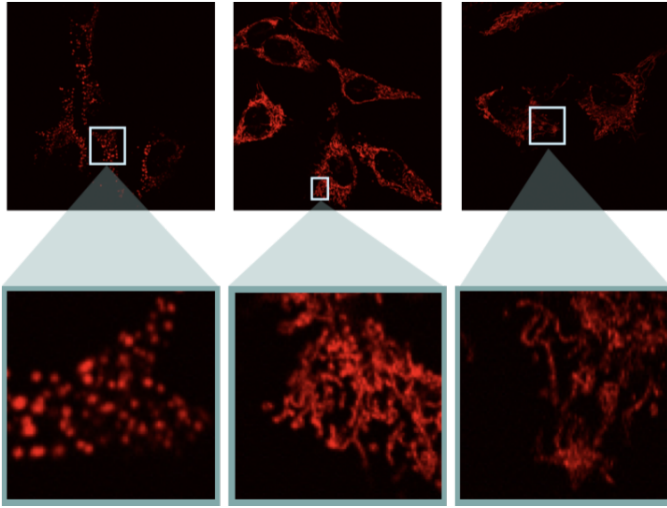


Fig. 2: Sample frames from each of the subsets of data. (Left) LLO induced mitochondrial fragmentation (Center) Wild type HeLa mitochondrial morphology (Right) Mdivi-1 induced mitochondrial hyperfusion

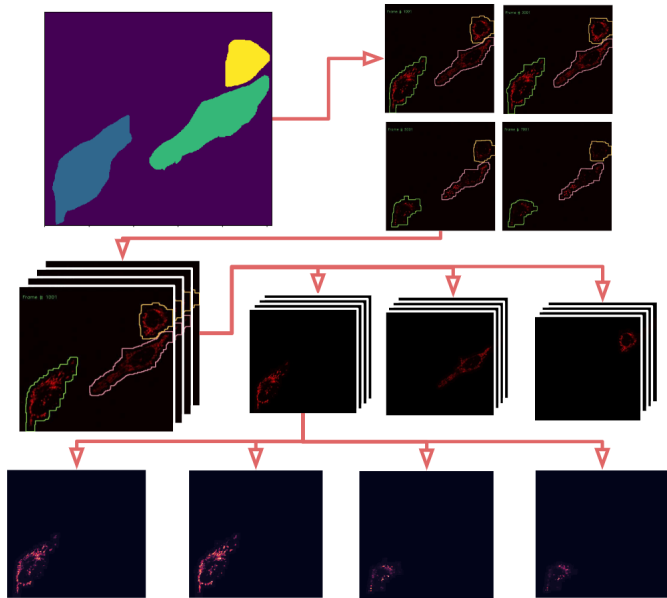


Fig. 3: Diagram of the cell segmentation process. (Top Left) Hand drawn masks of the first frame in VTK format were used to "seed" the deformable contours. (Top Right) A series of frames from a single video with autonomously drawn contours. (Middle) Stack of frames from a single video converted to separate videos for each cell. (Bottom) single cell video unraveled as grayscale image for frame by frame network modeling.

the ITK-SNAP software [YPCH⁺06] to label each cell manually in the first frame of each video, generating a VTK file with the segmentation maps (Fig. 3, top left).

Our segmentation process used these maps as "seeds", updating the maps at each frame of the video using deformable contours: iterative dilation, thresholding, and contour detection process over the entire video, resulting in a set of masks for each frame and each cell in the frame. These masks could then be used to pull out individual cells over the course of the video (Fig. 3).

While this process was very effective at following the cells, occasionally the model would lose small areas of mitochondrial

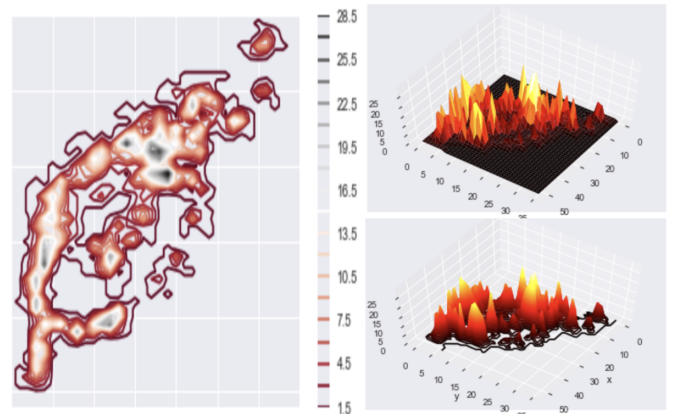


Fig. 4: (Left) a 2D probability representation of the intensity of a sample cell. (Top Right) the Intensity map of the image in a 3D representation. (Bottom Right) the 3D contour of the same cell.

mass which was sufficiently far away from the more contiguous structure. To compensate, we added a final process of iterative dilation to prevent loss and give a more generous contour. With these adjustments, we ran into a rare problem of cell contact or overlap. In response, we continued the iterative dilation with more iterations and smaller dilations checking for overlap with another map each iteration. In the case of an overlap, which would only be a few pixels with the small dilation kernel, we used a simple XOR to remove the few overlapping pixels while still allowing the mask to expand in areas unclaimed by other cells. With this case being rare, we found the process mostly followed any visible boundary of the adjacent cell.

The output of this step was the individual cell masks, one for each cell at each frame, providing a complete segmentation of each cell.

Social Network Engineering

To induce a network structure over the mitochondrial patterns of the segmented cells, we used a Gaussian Mixture Model (GMM). The means and covariances of the model components would represent two critical features of a social network: the individual nodes (means), and the nodes' relationships to each other (covariances). An independent model would be trained for each individual cell, and the model parameters would be permitted to evolve over the course of the videos to capture the changing underlying morphologies.

We first applied a Gaussian smoothing filter to minimize or eliminate artifacts in the video images. We then converted the frames of the video to a discrete probability distribution by normalizing the grayscale pixel intensities to sum to 1 (Fig. 4). Following the conversion to a probability density, we counted local pixel maxima and used these points--both the number of maxima found, and their spatial locations--as the initial components our GMM. These components were fed into the GMM $fit()$ procedure in scikit-learn (Fig. 5). The learned GMM components would minimize the disparity between the joint probability density of the GMM, and the original empirical probability density of the image, parameterizing the structure of the mitochondrial pattern. Using the learned components as nodes in the final network allow for the network structure to be learned purely from the mitochondrial topology.

The code for converting a single image frame to a discrete probability density function and learn the initial GMM components are as follows:

```
def img_to_px(image):
    """
    Converts the image to a probability
    distribution amenable to GMM.

    Parameters
    -----
    image : array, shape (H, W)
        8-bit grayscale image.

    Returns
    -----
    X : array, shape (N, 2)
    The data.
    """
    # We need the actual 2D coordinates of the
    # pixels.
    #The following is fairly standard practice for
    #generating a grid
    #of indices, often to evaluate some function on
    #a discrete surface.
    x = np.arange(image.shape[1])
    y = np.arange(image.shape[0])
    xx, yy = np.meshgrid(x, y)

    # Now we unroll the indices and stack them into
    #2D (i, j) coordinates.
    z = np.vstack([yy.flatten(), xx.flatten()]).T

    # Finally, we repeat each index by the number
    # of times of its pixel value.
    # That is our X--consider each pixel an
    #"event", and its value is the
    # number of times that event is observed.
    X = np.repeat(z, image.flatten(), axis = 0)
    return X

def skl_gmm(vid, vizual = False, skipframes = 10,
            threshold_abs = 6, min_distance = 10):
    """
    Runs a warm-start GMM over evenly-spaced
    frames of the video.

    Parameters
    -----
    vid : array, shape (f, x, y)
        Video, with f frames and spatial
        dimensions x by y.
    vizual : boolean
        True will show images and nodes
        (default: False).
    skipframes : integer
        Number of frames to skip (downsampling
        constant).

    Returns
    -----
    covars : array, shape (f, k, 2, 2)
        The k covariance matrices (each 2x2)
        for each of f frames.
    means : array, shape (f, k, 2)
        The k 2D means for each of f frames.
    """
    img = vid[0]
    if(vizual):
        plt.imshow(img)
        plt.show()
    X = image.img_to_px(img)
    PI, MU, CV = params.image_init(img, k = None,
        min_distance = min_distance,
        threshold_abs = threshold_abs)
    PR = np.array(list(map(sla.inv, CV)))
    gmmodel = GaussianMixture(n_components = CV.shape[0]
```

```
weights_init = PI, means_init = MU,
precisions_init = PR)
gmmodel.fit(X)
if(vizual):
    viz.plot_results(gmmodel.means_,
                    gmmodel.covariances_, 0, img.shape[1], 0,
                    img.shape[0], 0, 'this')

covars = [gmmodel.covariances_]
means = [gmmodel.means_]

#set warm start to true to use previous parameters
gmmodel.warm_start = True

for i in range(1+skipframes, vid.shape[0], skipframes):
    img = vid[i]
    if(vizual):
        plt.imshow(img)
        plt.show()

    X = image.img_to_px(img)
    gmmodel.fit(X)
    covars = np.append(covars,
                      [gmmodel.covariances_], axis = 0)
    means = np.append(means,
                     [gmmodel.means_], axis = 0)
    if(vizual):
        viz.plot_results(gmmodel.means_,
                        gmmodel.covariances_,0, img.shape[1],
                        0, img.shape[0], 0, 'this')

return means,covars
```

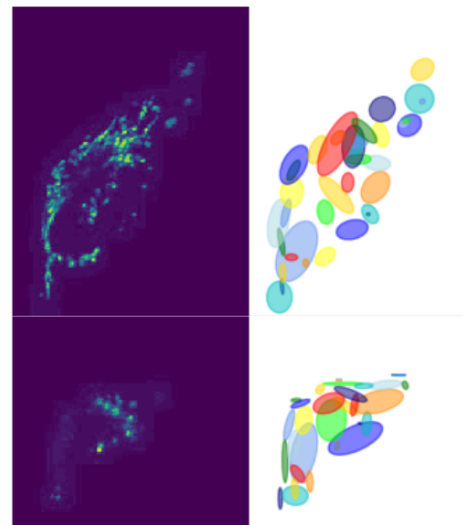


Fig. 5: A cell (Left) and the nodes (Right) as generated by a gaussian mixture model for the first (Rop) and last (Rottom) frames of a video showing a cell fragmented by LLO

For connecting the nodes with weighted edges, we explored multiple approaches that balanced realistically encapsulating the underlying biology (i.e., did not create connections between uncorrelated objects) and computational tractability. Initially, we chose a manual distance threshold and used this as the "neighborhood size" for the radial-basis function, a common connection-weighting metric that varies smoothly from 0 (not connected) to 1 (fully connected), and is a function of the Euclidean distance between the two nodes, weighted by the neighborhood size. A second attempt to make this process more data-driven was to replace the manually-crafted neighborhood size with the Gaussian covariance in the direction of the node to be connected (6, mid). In both cases, to avoid fully-connected graphs and induce some

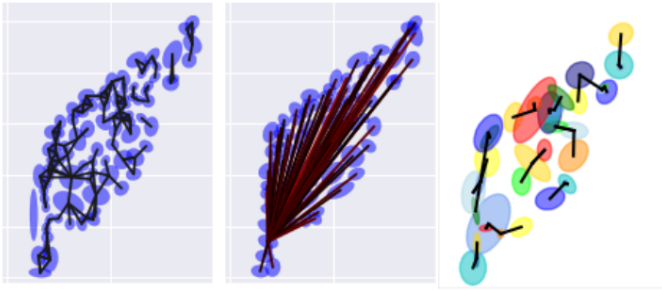


Fig. 6: (Left) A partially connected network with binary connections. (Center) A single node's weighted connection within a fully connected graph. (Right) A the strongest connection of each node as determined by our current affinity function

sparsity, we set a hard threshold on the maximum distance between nodes to connect (6, left).

While these produced networks with desirable properties, they did not fully reflect the underlying biology. Critically, the latter produced networks with connectivity levels that varied wildly even between subsequent frames of the same video. We interpreted this "thrashing" as noise: while we expect some systemic changes in the topology of the network through the formation and destruction of connections between nodes, we observed considerable shifts in this topology even in the control (wild-type) videos. Therefore, we sought a method for computing edge weights between nodes that was more robust to minor fluctuations in the underlying mitochondrial protein pattern. We also desired a similarity metric less dependent on Euclidean distance: this distance measure was entirely dependent on the magnification level of the microscope, an undesirable dependency and potential source of artifacts should the method be applied on data gathered from a variety of imaging modalities.

To address these shortcomings with determining network connectivity, we instead evaluated the Gaussian components directly and used that probability as the edge weight. This not only accounted for the anisotropy in the covariance of the Gaussian components, but also captured the asymmetry between components: by decoupling the direct link to Euclidean distance, the connections could instead be weighted by how probable the location of the node under consideration was (6, right). While this did result in an asymmetric graph matrix, it more accurately reflected the dynamics of the underlying biology, captured the relationships between nodes in a more intuitive metric, and was entirely data-driven with no hand-crafted thresholds.

Many popular social networks have asymmetric connections between users. For example, Twitter and Instagram permit users to follow another without being followed back. Even Facebook, which has a symmetric "friend" connection, has asymmetric underlying weights in terms of how friends interact over the network. Biologically speaking, there is little evidence to prefer a directed graph structure over an undirected one. However, with a cell's general Brownian behavior, the undirected structure seem more analogous and flexible; we would anticipate an empirical convergence to an undirected graph if the behavior warrants. This element of the our graph structure will be more cemented as we analyze the networks created.

To calculate our network structure in terms of the Gaussian components, we use the following functions (*normpdf* includes an implementation of a multivariate Gaussian probability density

function due to discrepancies with the *scipy.stats.norm* implementation):

```
def normpdf(X, mu, sigma):
    """
    Evaluates the PDF under the current GMM
    parameters.

    Parameters
    -----
    X : array, shape (N, d)
        The data.
    mu : array, shape (d,)
        Mean of the Gaussian.
    sigma : array, shape (d, d)
        Gaussian covariance.

    Returns
    -----
    px : array, shape (N,)
        The probability density of each data point,
        given the parameters.
    """
    d = 1 if len(X.shape) == 1 else X.shape[1]
    if d == 1:
        n = 1 / ((2 * np.pi * sigma) ** 0.5)
        e = np.exp(-((X - mu) ** 2) /
                    (2 * sigma))
        px = n * e
    else:
        det = sla.det(sigma)
        inv = sla.inv(sigma)
        p = np.einsum('ni,ji,ni->n', X - mu,
                    inv, X - mu)
        n = 1 / (((2 * np.pi) ** d) * det)
        ** 0.5)
        px = np.exp(-0.5 * p) * n
    return px

def aff_by_eval(means, covars):
    """
    finds an affinity table for a set of
    means and covariances representing nodes

    Parameters
    -----
    means : array, shape (k, 2)
        the list of means with k nodes
    covars : array, shape (k, 2, 2)
        the list of covars with k nodes

    Returns
    -----
    aff_Table : array, shape (k, k)

    """
    aff_Table = np.empty([means.shape[0], 0])
    for i, (mean, covar) in enumerate(zip(means,
        covars)):
        p_mus_Kx = normpdf(means, mean, covar)
        aff_Table = np.append(aff_Table,
            np.transpose([p_mus_Kx]), axis=1)
    return aff_Table

def get_all_aff_tables(means, covars):
    """
    finds all affinity table for a set of Frames
    each with lists of means and covariances

    Parameters
    -----
    means : array, shape (f, k, 2)
        the list of lists of means with f frames and
        k nodes
    covars : array, shape (k, 2, 2)
        the list of lists of covars with f frames
        with k nodes
    """
```



```

Returns
-----
aff_Table : array, shape (k, k)

"""
aff_Tables = [aff_by_eval(means[0], covars[0])]
for i in range(1, means.shape[0]):
    aff_Tables = np.append(aff_Tables,
        [aff_by_eval(means[i], covars[i])], axis = 0)
return aff_Tables

```

Current Insights and Future Work Discussion

After building networks using the described GMM method for each cell under varying conditions (control/wildtype, LLO, mdivi), we have qualitatively observed systemic differences in the learned model parameters that would separate these conditions. Interestingly, as the mitochondria fragment (i.e., LLO), the GMM components become more strongly connected, not less (7). We attribute this to a misinformed intuition: as the mitochondria fragment and the underlying probability density function becomes more uniform, the GMM components will likewise become more uniform, resulting in a more uniformly connected network. The overall number of connections also increases, as the cells tend to collapse at the same time as mitochondrial fragmentation, resulting in the same number of GMM components spatially colocating in a much smaller space, effectively "forcing" connections by virtue of proximity. By comparison, the control cell shows much less variation in the distribution of network connectivity and edge weights over time; this reflects a relatively stable social network, unperturbed by external stimuli.

The next step, then, is to develop a temporal model of the GMM component evolution in terms of the social network. This would take the form of a series of graph Laplacians and observing how the Laplacians change, likely as a function of Laplacian gradients. This would highlight specific portions of the social networks that covary over space and time; in other words, it would provide insight into the coordinated fragmentation or hyperfusion of the mitochondria in response to the provided stimulus. These features could then be incorporated into a broader supervised learning pipeline to distinguish patterns and discern the effects of an unknown stimulant (e.g., drug discovery), or an unsupervised learning pipeline to identify all observed mitochondrial phenotypes.

Additional methods of analyzing the graph structure of the social network would help to determine specific phenotypic changes induced by certain stimuli. In particular, classic graph metrics such as connectivity, cliques, and eigenvector centrality would help to precisely measure the global effects of certain stimuli on the mitochondria. Other algorithms, such as spectral clustering or PageRank for global network analysis from local phenomena would provide intuition into the local changes in mitochondrial phenotype responsible for inducing the global structure. These features would be invaluable for characterizing certain specific cell-wide or even organism-wide conditions.

We also aim to improve the process through which the social network is constructed in the first place. The incorporation of a single uniform component into the overall GMM would provide a robust method of accounting for background noise in the form of a learned, data-driven threshold. Additional refinements of the affinity function that determines the existence of connections between nodes, and their weight and direction, will be pursued: the Kullback-Leibler (KL) divergence is a popular method for

measuring the difference between two probability distributions, and would be a natural fit for evaluating how similar two GMM components are.

In this paper, we have presented a proof-of-concept for parameterizing and modeling spatiotemporal changes in diffuse subcellular protein patterns using GMMs. We have presented how the learned parameters of the GMM can be updated to account for changing biological phenotypes, and how these parameters can then be used to induce a social network of interacting nodes. Finally, we show how the properties of the social network can be interpreted to provide biological insights, in particular how the underlying system may be responding to some kind of stimulus. This has potential implications in fundamental biology and translational biomedicine; we aim to complete our analysis package and release it as open source for the research community to use in the near future.

Acknowledgments

This project was supported in part by a grant from the National Science Foundation (#1458766).

REFERENCES

- [CSCI+08] Ann Cassidy-Stone, Jerry E Chipuk, Elena Ingerman, Cheng Song, Choong Yoo, Tomomi Kuwana, Mark J Kurth, Jared T Shaw, Jenny E Hinshaw, Douglas R Green, et al. Chemical inhibition of the mitochondrial division dynamin reveals its role in bax/bak-dependent mitochondrial outer membrane permeabilization. *Developmental cell*, 14(2):193–204, 2008.
- [DGD+16] Andrew T Drysdale, Logan Grosenick, Jonathan Downar, Katharine Dunlop, Farrokh Mansouri, Yue Meng, Robert N Fetcho, Benjamin Zebley, Desmond J Oathes, Amit Etkin, Alan F Schatzberg, Keith Sudheimer, Jennifer Keller, Helen S Mayberg, Faith M Gunning, George S Alexopoulos, Michael D Fox, Alvaro Pascual-Leone, Henning U Voss, BJ Casey, Marc J Dubin, and Conor Liston. Resting-state connectivity biomarkers define neurophysiological subtypes of depression. *Nature Medicine*, 2016. URL: <http://dx.doi.org/10.1038/nm.4246>, doi:10.1038/nm.4246.
- [FCGQR15] Kari Fine-Coulson, Steeve Giguère, Frederick D Quinn, and Barbara J Reaves. Infection of a549 human type ii epithelial cells with mycobacterium tuberculosis induces changes in mitochondrial morphology, distribution and mass that are dependent on the early secreted antigen, esat-6. *Microbes and infection*, 17(10):689–697, 2015.
- [KPSBW08] Andrew B Knott, Guy Perkins, Robert Schwarzenbacher, and Ella Bossy-Wetzel. Mitochondrial fragmentation in neurodegeneration. *Nature Reviews Neuroscience*, 9(7):505, 2008.
- [SBS+11] Fabrizia Stavru, Frédéric Bouillaud, Anna Sartori, Daniel Ricquier, and Pascale Cossart. *Listeria monocytogenes* transiently alters mitochondrial dynamics during infection. *Proceedings of the National Academy of Sciences*, 108(9):3612–3617, 2011.
- [STS17] Valentine Svensson, Sarah A. Teichmann, and Oliver Stegle. Spatialde - identification of spatially variable genes. *bioRxiv*, 2017. URL: <https://www.biorxiv.org/content/early/2017/11/08/143321>, arXiv:<https://www.biorxiv.org/content/early/2017/11/08/143321.full.pdf>, doi:10.1101/143321.
- [VCL+17] Alex M Valm, Sarah Cohen, Wesley R Legant, Justin Melunis, Uri Hershberg, Eric Wait, Andrew R Cohen, Michael W Davidson, Eric Betzig, and Jennifer Lippincott-Schwartz. Applying systems-level spectral imaging and analysis to reveal the organelle interactome. *Nature*, 546(7656):162, 2017.
- [WL16] Timothy Wai and Thomas Langer. Mitochondrial dynamics and metabolic regulation. *Trends in Endocrinology & Metabolism*, 27(2):105–117, 2016.
- [YPCH+06] Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. *Neuroimage*, 31(3):1116–1128, 2006.

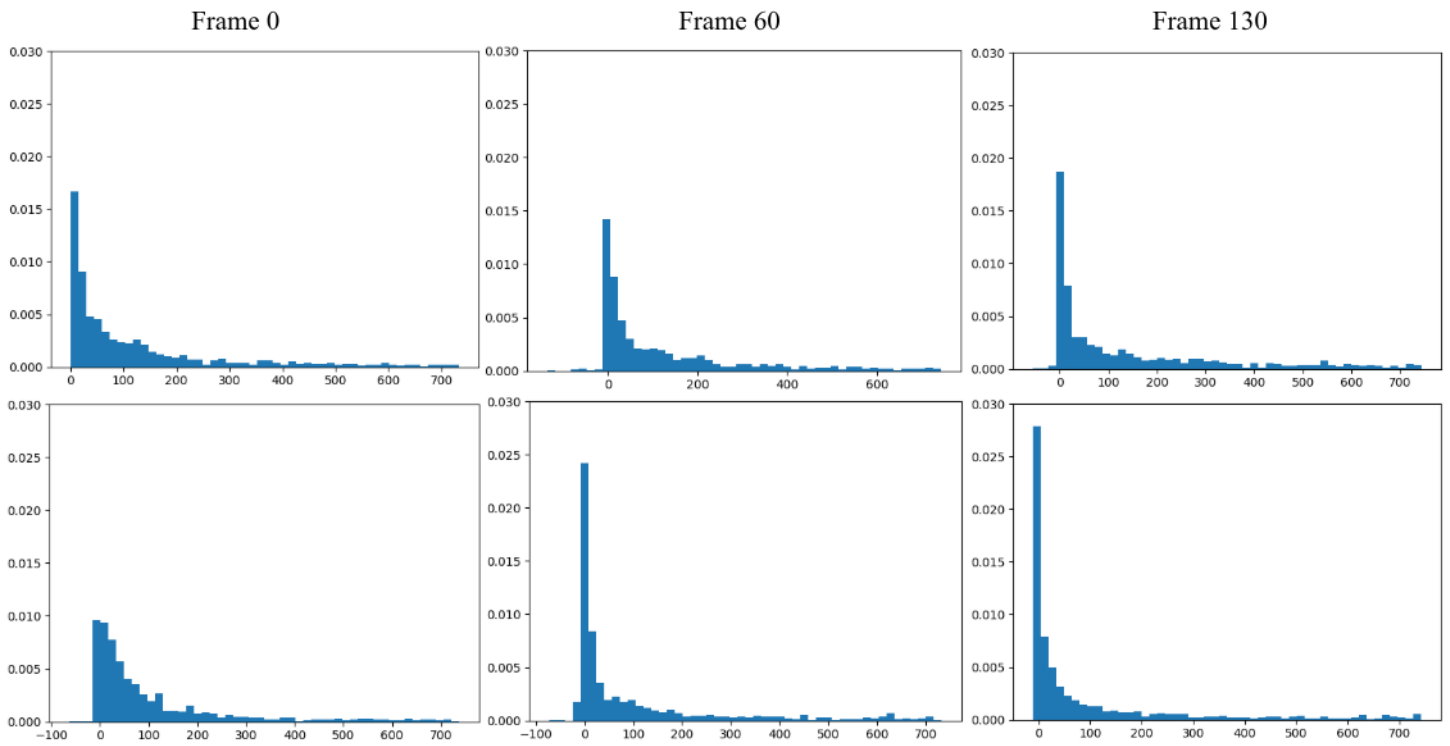


Fig. 7: A series of distribution plots of the negative log of values found in six affinity tables developed using the model learned at an early, middle, and late video frame. (Top) The tables generated from a control cell which show little variation in distribution. (Bottom) The tables generated from the LLO cell which shows a drastic increase in connectivity over time as the cell fragments.

[ZLN13] Jian Zhao, Urban Lendahl, and Monica Nistér. Regulation of mitochondrial dynamics: convergences and divergences between yeast and vertebrates. *Cellular and molecular life sciences*, 70(6):951–976, 2013.

Cloudknot: A Python Library to Run your Existing Code on AWS Batch

Adam Richie-Halford^{‡*}, Ariel Rokem[‡]

<https://youtu.be/D9LPzqoZ3f8>



Abstract—We introduce Cloudknot, a software library that simplifies cloud-based distributed computing by programmatically executing user-defined functions (UDFs) in AWS Batch. It takes as input a Python function, packages it as a container, creates all the necessary AWS constituent resources to submit jobs, monitors their execution and gathers the results, all from within the Python environment. Cloudknot minimizes the cognitive load of learning a new API by introducing only one new object and using the familiar `map` method. It overcomes limitations of previous similar libraries, such as Pywren, that runs UDFs on AWS Lambda, because most data science workloads exceed the current limits of AWS Lambda on execution time, RAM, and local storage.

Index Terms—Cloud computing, Amazon Web Services, Distributed computing

Introduction

In the quest to minimize time-to-first-result, data scientists are increasingly turning to cloud-based distributed computing with commercial vendors like Amazon Web Services (AWS). Cloud computing platforms have the advantage of linear scalability: users can access limitless computing resources to meet the demands of their computational workloads. At the same time they offer elasticity: resources are provisioned as-needed and can be decommissioned when they are no longer needed. In data-intensive research scenarios in which large computational workloads are coupled with large amounts of data this could, in principle, offer substantial speedups.

But the complexity and learning curve associated with a transition to cloud computing make it inaccessible to beginners. This transition cost has been improving. For example, Dask [Roc15] used to be difficult to run in parallel in a cloud computing environment, but it is now more accessible, thanks in part to tools such as `dask-ec2` [Rod17] and `kubernetes/helm` [Aut18]. Yet despite these improvements, computation in the cloud remains inaccessible to many researchers who have not had previous exposure to distributed computing.

A number of Python libraries have sought to close this gap by allowing users to interact seamlessly with AWS resources from within their Python environment. For example, Cottoncandy allows users to store and access numpy array data on Amazon S3 [NEZH⁺17]. Pywren [JPV⁺17] enables users to run their

existing Python code on AWS Lambda, providing convenient distributed execution for jobs that fall within the limits of this service¹. However, these limitations are impractical for many data-oriented workloads, which require more RAM and local storage, longer compute times, and complex dependencies. The AWS Batch service offers a platform for workloads with these requirements. Batch dynamically provisions AWS resources based on the volume and requirements of user-submitted jobs. Instead of provisioning and managing their own batch computing jobs, users specify job constraints, such as the amount of memory required for a single job, and the number of jobs. AWS Batch manages the job distribution to satisfy those constraints. The user can optionally constrain the cost by using Amazon EC2 Spot Instances [AWS18a] and specifying a bid percentage².

One of the main advantages of Batch, relative to the provisioning of your own compute instances is that it abstracts away the exact details of the infrastructure that is needed, offering instead relatively straight-forward abstractions:

- a *job*, which is an atomic, independent task to repeat on multiple inputs, encapsulated in a linux executable, a bash script or a Docker container;
- a *job definition*, which connects the job with the compute resources it require;
- a *compute environment*, which defines the configuration of the computational resources needed, such as number of processors, or amount of RAM;
- a *job queue*, where jobs reside until they are run in a compute environment.

While Batch provides useful functional abstractions for processing data in bulk, the user interface provided through the AWS web console still resists automation, requires learning many of the terms that control its execution and does not facilitate scripting and/or reproducibility [AWS18b]. The AWS Python API offers a programming interface that can control the execution of computational tasks in AWS Batch, but it is not currently designed to offer an accessible single point of access to these resources.

Here, we introduce a new Python library with support for Python 2.7 and 3.5+: Cloudknot [RHR18a] [RHR18c], that launches Python functions as jobs on the AWS Batch service, thereby lifting these limitations. Rather than introducing its own

* Corresponding author: richiehalford@gmail.com

‡ University of Washington, Seattle, WA

1. Current limits include a maximum of 300 seconds of execution time, 1.5 GB of RAM, 512 MB of local storage, and no root access.

2. The bid percentage is the maximum price, expressed as a percentage of the on-demand EC2 instance price, with which to bid on unused EC2 capacity.

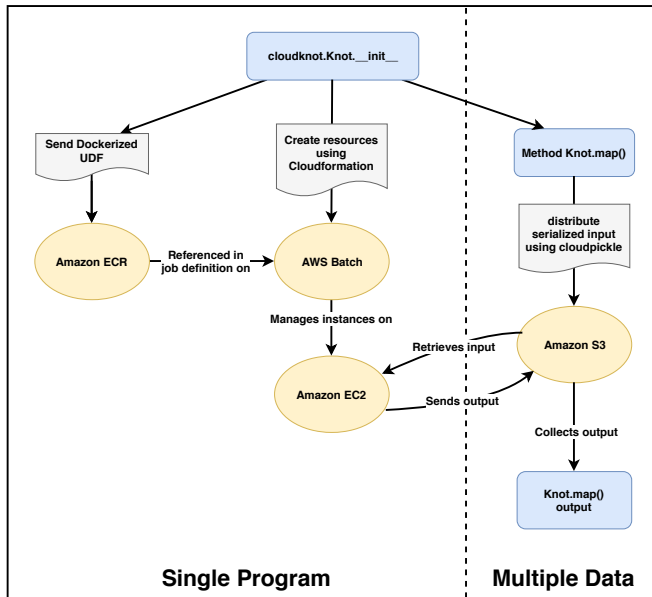


Fig. 1: Cloudknot’s SPMD workflow. The left two columns depict steps Cloudknot takes to create the single program (SP). The right column depicts Cloudknot’s management of the multiple data (MD). Blue rounded squares represent components of Cloudknot’s user-facing API. Yellow circles represent AWS resources. Grey document shapes represent containers, templates, or data used to communicate with cloud resources.

set of terms and abstractions, Cloudknot provides a simple abstraction on top of `Executor` objects whose results are returned by concurrent futures. Users of Cloudknot have to familiarize themselves with one new object: the `Knot`. While some of its functionality will initially be new to users of Cloudknot (e.g., the way that resources on AWS are managed), its `map` method should be familiar to most Python users.

The next section discusses Cloudknot’s approach to parallelism and the API section describes Cloudknot’s user interface. In the Examples section, we demonstrate a few of Cloudknot’s use cases, including examples with data ranging from hundreds of GB to several TB. We then summarize the trade-offs between performance and accessibility in the Conclusion.

Design

The primary object in Cloudknot is the `Knot`, which employs the single program, multiple data (SPMD) paradigm to achieve parallelism. In this section, we describe Cloudknot’s approach to establishing the single program (SP) and managing the multiple data (MD). `Knot`’s user-facing API and interactions with cloud-based resources are depicted in Figure 1.

Single Program (SP)

The `Knot` object creates the single program on initialization, taking a user-defined function (UDF) as input and wrapping it in a command line interface (CLI), which downloads data from an Amazon Simple Storage Service (S3) bucket specified by an input URL. The UDF is also wrapped in a Python decorator that sends its output back to an S3 bucket. So in total, the resulting command line program downloads input data from S3, executes the UDF, and sends output back to S3. `Knot` then packages the CLI, along with its dependencies, into a Docker container. The

container is uploaded into the Amazon Elastic Container Registry (ECR). Cloudknot’s use of Docker allows it to handle non-trivial software and data dependencies (see examples below). This is because Docker provides a consistent and isolated environment, allowing complete control over the software dependencies of a particular application, and near-immediate deployment of these dependencies [Boe14].

Separately, `Knot` uses an AWS CloudFormation template to create the AWS resources required by AWS Batch³. `Knot` passes the location of the Docker container on AWS ECR to its job definition so that all jobs execute the SP. The user may restrict the compute environment of the `Knot` to only certain instance types (e.g. `c4.2xlarge`) or may choose a specific Amazon Machine Image (AMI) to be loaded on each compute resource. Or, they may simply request a minimum, desired, and maximum number of virtual CPUs and let AWS Batch select and manage the EC2 instances.

`Knot` uses job definition and compute environment defaults that are conservative enough to run most simple jobs, with the goal of minimizing errors due to insufficient resources. The casual user may never need to concern themselves with selecting an instance type or specifying an AMI. Users who want to minimize costs by specifying the minimum sufficient resources or users who need additional resources for intensive jobs can control their jobs’ memory requirements, instance types, or AMIs. This might be necessary if the jobs require special hardware (e.g. GPGPU computing) or if the user wants more fine-grained control over which resources are launched.

One of the most complex aspects of AWS is its permissions model⁴. Here, we assume that the user has the permissions needed to run AWS Batch in the console. We also provide users with the minimal necessary permissions in the documentation.

Finally, `Knot` exposes AWS resource tags [AWS18c] to the user, allowing the user to assign metadata key-value pairs to each created resource. This facilitates management of Cloudknot generated resources and allows the user to quickly recognize Cloudknot resources in the AWS console.

Multiple Data (MD)

To operate on the MD, the `Knot.map()` method uses a simple for loop to iterate over the outer-most dimension of the input array and assign each element to a separate AWS Batch job.

3. The required resources are

- AWS Identity and Access Management (IAM) Roles
 - a batch service IAM role to allow AWS Batch to make calls to other AWS services on the user’s behalf;
 - an Elastic Container Service (ECS) instance role to be attached to each container instance when it is launched;
 - an Elastic Cloud Compute (EC2) Spot Fleet role to allow Spot Fleet to bid on, launch, and terminate instances if the user chooses to use Spot Fleet instances instead of dedicated EC2 instances;
- an AWS Virtual Private Cloud (VPC) with subnets and a security group;
- an AWS Batch job definition specifying the job to be run;
- an AWS Batch job queue that holds jobs until scheduled into a compute environment;
- and an AWS Batch compute environment, which is a set of compute resources that will be used to run jobs.

4. <https://docs.aws.amazon.com/IAM/latest/UserGuide>

The Knot serializes each element in the array and sends it to S3, organizing the data in a schema that is internally consistent with the expectations of the CLI. It then launches an AWS Batch array job (or optionally, separate individual Batch jobs) to execute the program over these data. When run, each batch job selects its own input, executes the UDF, and returns its serialized output to S3.

If the instances and S3 bucket are in the same region, then users do not pay for transfer from S3 to the EC2 instances and back. They pay only for transfer out of the data center (i.e. from their local machine to S3 and back). Transfer speed within the data center also outperforms transfer speed between data centers. So it is both less costly and more performant to colocate the Cloudknot S3 bucket with the EC2 instances. Cloudknot includes utility functions to change regions and S3 buckets for this purpose.

In the last step, `Knot.map()` downloads the output from S3 and returns it to the user. Since AWS Batch allows arbitrarily long execution times, `Knot.map()` returns a list of futures for the results, mimicking Python's concurrent futures' `Executor` objects. If the results are too large to fit on the local machine, the user may augment their UDF to write results to S3 or some other remote storage and then simply return the address at which to retrieve the result.

Under the hood, `Knot.map()` creates a `concurrent.futures.ThreadPoolExecutor` instance where each thread intermittently queries S3 for its returned output. The results are encapsulated in `concurrent.futures.Future` objects, allowing asynchronous execution. The user can use `Future` methods such as `done()` and `result()` to test for success or view the results. This also allows attaching callbacks to the results using the `add_done_callback()` method. For example a user may want to perform a local reduction on results generated on AWS Batch.

API

The above interactions with AWS resources are hidden from the user. The advanced and/or curious user can customize the Docker container or CloudFormation template. But for most use cases, the user interacts only with the `Knot` object. This section provides an example calculating the value of π as a pedagogical introduction to the Cloudknot API.

We first import Cloudknot and define the function that we would like to run on AWS Batch. Cloudknot uses the `pipreqs` [Kra17] package to generate the requirements file used to install dependencies in the Docker container on AWS ECR. So all required packages must be imported in the source code of the UDF itself.

```
import cloudknot as ck

def monte_pi_count(n):
    import numpy as np
    x = np.random.rand(n)
    y = np.random.rand(n)
    return np.count_nonzero(x * x + y * y <= 1.0)
```

Next, we create a `Knot` instance and pass the UDF using the `func` argument. The `name` argument affects the names of resources created on AWS. For example, in this case, the created job definition would be named `pi-calc-cloudknot-job-definition`:

```
knot = ck.Knot(name='pi-calc', func=monte_pi_count)
```

We submit jobs with the `Knot.map()` method:

```
import numpy as np # for np.ones
n_jobs, n_samples = 1000, 100000000
args = np.ones(n_jobs, dtype=np.int32) * n_samples
future = knot.map(args)
```

This will launch an AWS Batch array job with 20 child jobs, one for each element of the input array. Cloudknot can accommodate functions with multiple inputs by passing the `map()` method a sequence of tuples of input arguments and the `starmap=True` argument. For example, if the UDF signature were `def udf(arg0, arg1)`, one could execute `udf` over all combinations of `arg0` in `[1, 2, 3]` and `arg1` in `['a', 'b', 'c']` by calling

```
args = list(itertools.product([1, 2, 3],
                              ['a', 'b', 'c']))
future = knot.map(args, starmap=True)
```

We can then query the result status using `future.done()` and retrieve the results using `future.result()`, which will block until results are returned unless the user passes an optional `timeout` argument. We can also check the status of all the jobs that have been submitted with this `Knot` instance by inspecting the `knot.jobs` property, which returns a list of `cloudknot.BatchJob` instances, each of which has its own `done` property and `result()` method. So in the example above, `future.done()` is equivalent to `knot.jobs[-1].done` and `future.result()` is equivalent to `knot.jobs[-1].result()`. In this way, users have access to AWS Batch job results that they have run in past sessions.

In this pedagogical example, we are estimating π using the Monte Carlo method. `Knot.map()` returns a future for an array of counts of random points that fall within the circle enclosed by the unit square. To get the final estimate of π , we need to sum all the elements of this array and divide by four, a simple use case for `future.add_done_callback()`:

```
PI = 0.0
n_total = n_samples * n_jobs
def pi_from_future(future):
    global PI
    PI = 4.0 * np.sum(future.result()) / n_total
future.add_done_callback(pi_from_future)
```

Lastly, without navigating to the AWS console, we can get a quick summary of the status of all jobs submitted with this `Knot` using

```
>>> knot.view_jobs()
Job ID      Name      Status
-----
fcd2a14b... pi-calc-0  PENDING
```

Examples

In this section, we will present a few use cases of Cloudknot. We will start with examples that have minimal software and data dependencies, and increase the complexity by adding first data dependencies and subsequently complex software and resource dependencies. These and other examples are available in Jupyter Notebooks in the Cloudknot repository [RHR18b].

Solving differential equations

Simulations executed with Cloudknot do not have to comply with any particular memory or time limitations. This is in contrast

to Pywren’s limitations, which stem from the use of the AWS Lambda service. On the other hand, Cloudknot’s use of AWS Batch increases the overhead associated with creating AWS resources and uploading a Docker container to ECR. While this infrastructure setup time can be minimized by reusing AWS resources that were created in a previous session, this setup time suits use-cases for which execution time is much greater than the time required to create the necessary resources on AWS.

To demonstrate this, we used Cloudknot and Pywren to find the steady-state solution to the two-dimensional heat equation by the Gauss-Seidel method [BBC+94]. The method chosen is suboptimal, as is the specific implementation of the method, and serves only as a benchmarking tool. In this unrealistic example, we wish to parallelize execution both over a range of different boundary conditions and over a range of grid sizes.

First, we hold the grid size constant at 10×10 and parallelize over different temperature constraints on one edge of the simulation grid. We investigate the scaling of job execution time as a function of the size of the argument array. In Figure 2 we show the execution time as a function of n_{args} , the length of the argument array (with both on \log_2 scales). We tested scaling using Cloudknot’s default parameters and also using custom parameters⁵. Regardless of the `KnOt` parameters, Pywren outperformed Cloudknot at all argument array sizes. Indeed, Pywren appears to achieve constant scaling between $2^2 \leq n_{\text{args}} \leq 2^9$, revealing AWS Lambda’s capabilities for massively parallel computation. For $n_{\text{args}} > 2^9$, Pywren appears to conform to linear scaling with a constant of roughly 0.25. By contrast, Cloudknot exhibits noisy linear scaling for $n_{\text{args}} \gtrsim 2^5$, with constants of roughly 2 for the custom configuration and roughly 4 for the default configuration. Precise determination of these scaling constants would require more data for a larger range of argument sizes.

For the data in Figure 3, we still parallelized over only five different temperature constraints, but we did so for increasing grid sizes. Grid sizes beyond 125×125 required an individual job execution time that exceeded the AWS Lambda execution limit of 300s. So Pywren was unable to compute on the larger grid sizes. There is a crossover point around 80×80 where Cloudknot outperforms Pywren. Before this point, AWS Lambda’s fast triggering and continuous scaling surpass the AWS Batch queueing system. Conversely, past this point the compute power of each individual EC2 instance launched by AWS Batch is enough to compensate for the difference in queueing performance.

Taken together, Figures 2 and 3 indicate that if a UDF can be executed within AWS Lambda’s five minute execution time and 1.5 GB memory limitations and does not have software and data dependencies that would prohibit using Pywren, it should be parallelized on AWS using Pywren rather than Cloudknot. However, when simulations are too large or complicated to fit well into Pywren’s framework, Cloudknot is the appropriate tool to simplify their distributed execution on AWS. Pywren’s authors note that the AWS Lambda limits are not fixed and are likely to improve. We agree and note only that EC2 and AWS Batch limitations are likely to improve as well. So long as there exists a computational regime between the two sets of limitations,

⁵ Default settings are `min_vcpus=0`, `desired_vcpus=8`, and `max_vcpus=256`. Custom settings are `desired_vcpus=2048`, `max_vcpus=4096`, and `min_vcpus=512`. Both default and custom Cloudknot cases were also limited by the EC2 service limits for our region and account, which vary by instance type but never exceeded 200 instances.

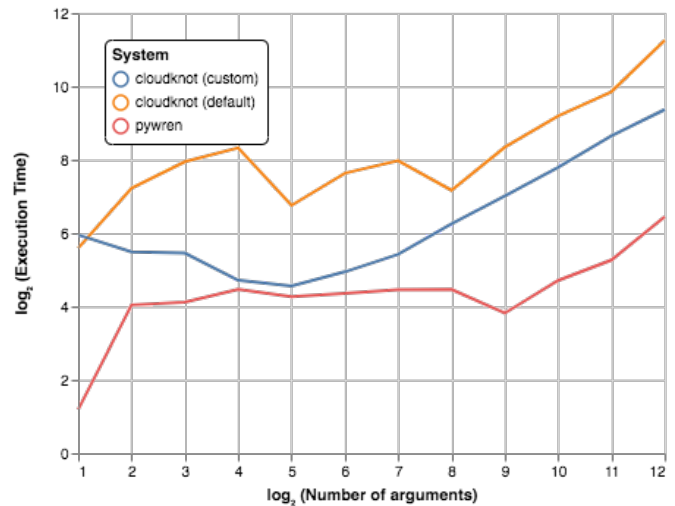


Fig. 2: Execution time to find solutions of the 2D heat equation for many different temperature constraints on a 10×10 grid. We show execution time scaling as a function of the number of constraints for Pywren, the default Cloudknot configuration, and a Cloudknot configuration with more available vCPUs. Pywren outperforms Cloudknot in all cases. We posit that the additional overhead associated with building the Docker image, along with EC2 service limits affected Cloudknot’s throughput.

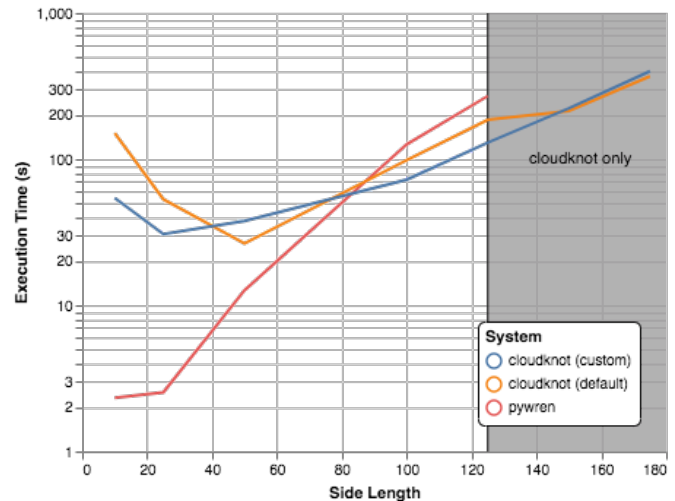


Fig. 3: Execution time to find five solutions to the 2D heat equation as a function of grid size. Grid sizes above 125×125 exceed Pywren’s limit on execution time of 300 sec. The cross-over point at around 80×80 occurs when it is more beneficial to have the more powerful EC2 instances provided by Cloudknot with AWS Batch than the massively parallel execution provided by Pywren with AWS Lambda.

Cloudknot can offer researchers a simple platform with which to execute their scientific workloads.

Data Dependencies: Analysis of magnetic resonance imaging data

Because Cloudknot is run on the standard AWS infrastructure, it allows specification of complex and large data dependencies. Dependency of individual tasks on data can be addressed by preloading the data into object storage on S3, and then downloading of individual bits of data needed to complete each task into the individual worker machines.

As an example, we implemented a pipeline for analysis of human MRI data. Human MRI data is a good use-case for a system such as Cloudknot because much of the analysis proceeds in a parallel manner. Even for large datasets with multiple subjects, a large part of the analysis is conducted first at the level of each individual brain. Aggregation of information across brains is typically done after many preprocessing and analysis stages at the level of each individual subject.

For example, diffusion MRI (dMRI) is a method that measures the properties of the connections between different regions of the brain. Over the last few decades, this method has been used to establish the role of these connections in many different cognitive and behavioral properties of the human brain, and to delineate the role that the biology of these connections plays in neurological and psychiatric disorders [Wan16]. Because of the interest in these connections, several large consortium efforts for data collection have aggregated large datasets of human dMRI data from multiple different subjects [GSM⁺16].

In the analysis of dMRI data, the first few steps are done at the individual level. For example, the selection of regions of interest within each image and the denoising and initial modeling of the data can all be completed at the individual level in parallel. In a previous study, we implemented a dMRI analysis pipeline that contained these steps and we used it to compare several Big Data systems as a basis for efficient scientific image processing [MDZ⁺17]. Here, we reused this pipeline. This allows us to compare the performance of Cloudknot directly against the performance of several alternative systems for distributed computing that were studied in our previous work: Spark [ZCF⁺10], Myria [HTdAC⁺14] and Dask [Roc15].

In Cloudknot, we used the reference implementation from this previous study written in Python and using methods from Dipy [GBA⁺14], which are implemented in Python and Cython. In contrast to the other systems, essentially no changes had to be made to the reference implementation when using Cloudknot, except to download the part of the data required for an individual job from S3 into the individual instances. Parallelization was implemented only at the level of individual subjects, and a naive serial approach was taken at the level of each individual.

We found that with a small number of subjects this reference implementation is significantly slower with Cloudknot compared to the parallelized implementation in these other systems. But the relative advantage of these systems diminishes substantially as the number of subjects grows larger (Figure 4), and the benefits of parallelization across subjects starts to be more substantial. With the largest number of subjects used, Cloudknot processed 25 subjects 10% slower than Spark and Myria; however, it was 25% slower than Dask, the fastest of the tools that we previously benchmarked.

There are two important caveats to this analysis: the first is that the analysis with the other systems was conducted on a cluster with a fixed allocation of 16 nodes (each node was an AWS r3.2xlarge instance with 8 vCPUs). The benchmark code does run faster with more nodes added to the cluster [MDZ⁺17]. The largest amount of data that was benchmarked was for 25 subjects, corresponding to 105 GB of input data and a maximum of 210 GB of intermediate data. Notably, even for this amount of data, Cloudknot deployed only two instances of the r4.16xlarge type -- each with 64 vCPUs and 488 GB of RAM. In terms of RAM, this is the equivalent of a 16 node cluster of r3.2xlarge instances, but the number of CPUs deployed to the task is about half. In

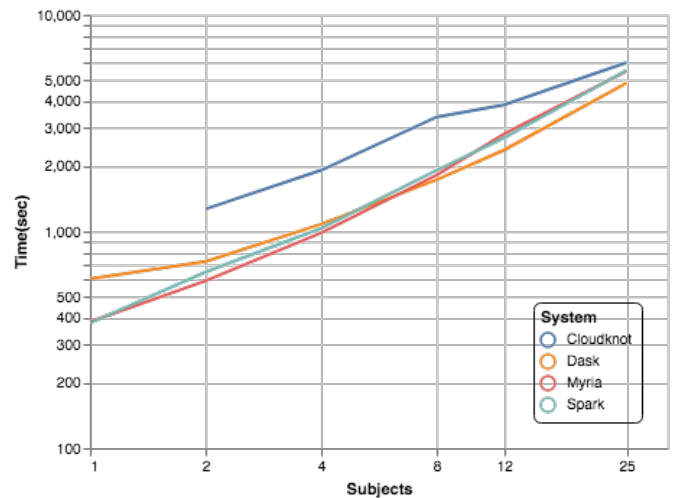


Fig. 4: MRI analysis pipeline with data requirements. A comparison of Cloudknot performance to other parallel computing systems: Dask, Spark and Myria, based on a previous benchmark [MDZ⁺17]. Cloudknot is orders of magnitude slower for small amounts of data, but reaches within 10-25 % of these systems' performance for large amounts of data.

general, users can choose to scale vertically (i.e., larger instance types, with more CPUs) or horizontally (i.e., more machines of smaller instance types) through the `instance_types` keyword argument to `Knot`. Additional scaling can also be reached by expanding the cluster with `min_vcpus`. The second caveat to these results is that the comparison timing data for the other systems is from early 2017, and these systems may have evolved and improved since.

Data and software dependencies: analysis of microscopy data

The MRI example demonstrates the use of a large and rather complex dataset. In addition, Cloudknot can manage complex software dependencies. Researchers in cell biology, molecular engineering and nano-engineering are also increasingly relying on methods that generate large amounts of data and on analysis that requires large amounts of computing power. For example, in experiments that evaluate the mobility of synthetically designed nano-particles in biological tissue [Nan17], [NWS⁺12], researchers may record movies of microscopic images of the tissue at high spatial and temporal resolution and with a wide field of view, resulting in large amounts of image data, often stored in multiple large files. These collections often reach several TB in size.

To analyze these experiments, researchers rely on software implemented in ImageJ for particle segmentation and tracking, such as TrackMate [TPS⁺17]. However, when applied to large amounts of data, using TrackMate serially in each experiment can be prohibitively time consuming. One solution is to divide the movies spatially into smaller field of view movies, and analyze them in parallel.

ImageJ and Trackmate are written in Java and can be scripted using Jython. This implies complex software dependencies, because the software requires installation of the ImageJ Jython runtime. Because Cloudknot relies on docker, this installation can be managed using the command line interface (i.e., `wget`). Once a docker image is created that contains the software dependencies for a particular analysis, Python code can be written on top of it to

execute system calls that will run the analysis. This approach was recently implemented in [Cur18].

Additional complexity in this use-case is caused by the volume of data. Because of the data size in this case, a custom AMI had to be created from the AWS Batch AMI, that includes a larger volume (Batch AMI volumes are limited to 30 GB of disk-space).

Conclusion

Cloudknot simplifies cloud-based distributed computing by programmatically executing UDFs in AWS Batch. This lowers the barrier to cloud computing and allows users to launch massive workloads at scale from within their Python environment.

We have demonstrated Cloudknot's ability to execute complex algorithms over vast quantities of data using real-world examples from neuroimaging and microscopy. And we've included analyses that show Cloudknot's performance compared to other distributed computing frameworks. On one hand, scaling charts like the ones in Figures 2, 3, and 4 are important because they show potential users the relative cost in execution time of using Cloudknot compared to other distributed computing platforms.

On the other hand, the timing results in this paper, indeed most benchmark results in general, measure the bare execution time, capturing only partial information about the time that it takes to reach a computational result. This is because all the distributed systems currently available require some amount of systems administration and often incur non-trivial setup time. In addition, most of the existing systems currently require some amount of rewriting of the original code [MDZ⁺17]. If the amount of time that a user will spend learning a new queuing system or batch processing language, administering this system, and rewriting their code for this system exceeds the time savings due to reduced execution time, then it will be advantageous to accept Cloudknot's suboptimal execution time in order to use its simplified API. Once they gain access to AWS Batch, beginning Cloudknot users simply add an extra import statement, instantiate a `Knot` object, call the `map()` method, and wait for results. And because Cloudknot is built using Docker and the AWS Batch infrastructure, it can accommodate the needs of more advanced users who want to augment their Docker files or specify instance types.

Cloudknot trades runtime performance for development performance and is best used when development speed matters most. Its simple API makes it a viable tool for researchers who want distributed execution of their computational workflow, from within their Python environment, without the steep learning curve of learning a new platform. It may have business applications as well since data scientists performing exploratory analysis would benefit from short development times.

Future Work

Cloudknot can benefit from several enhancements:

- In future developments, we will focus our attention on domain-specific applications (in neuroimaging, for example) and include enhancements and bug-fixes that arise from use in our own research.
- Unlike Dask, Cloudknot does not support computational pipelines that define dependencies between different tasks. Future releases may support job dependencies so that specific jobs can be scheduled to wait for the results of previously submitted jobs.

- Cloudknot could also provide a simple way to connect to EC2 instances to allow in-situ monitoring of running jobs. To do this now, a user must look up an EC2 instance's address in the AWS console and connect to that instance using an SSH client. Future releases may launch this SSH terminal from within the Python session.
- Knot uses hard-coded defaults for the configuration of its job definition and compute environment. Future Cloudknot releases could intelligently estimate these defaults based on the UDF and the input data. For example, Knot could estimate its resource requirements by executing the UDF on one element of the input array many times using a variety of EC2 instance types. By recording the execution time, memory consumption, and disk usage for each trial, Knot could then adopt the configuration parameters of the best⁶ run and apply those to the remaining input.

In addition to these capability enhancements, Cloudknot could benefit from performance enhancements designed to address the performance gap with other distributed computing platforms. This might involve prebuilding certain Docker containers or intelligently selecting an AWS region to minimize cost or queueing time. Lastly, we claimed that Cloudknot's simple API likely gives it a gentler learning curve than other distributed computing platforms, but we did not rigorously compare the time investment required to learn how to use Cloudknot, relative to other systems. Future work may seek to fill this gap with a comparative human-computer interaction (HCI) study.

Acknowledgements

This work was funded through a grant from the Gordon & Betty Moore Foundation and the Alfred P. Sloan Foundation to the University of Washington eScience Institute. Thanks to Chad Curtis and Elizabeth Nance for the collaboration on the implementation of a Cloudknot pipeline for analysis of microscopy data.

REFERENCES

- [Aut18] The Kubernetes Authors. Helm: The package manager for kubernetes. <https://helm.sh/>, 2018.
- [AWS18a] Inc. Amazon Web Services. Amazon ec2 spot instances. <https://aws.amazon.com/ec2/spot>, 2018.
- [AWS18b] Inc. Amazon Web Services. Getting started with aws batch. https://docs.aws.amazon.com/batch/latest/userguide/Batch_GetStarted.html, 2018.
- [AWS18c] Inc. Amazon Web Services. Tagging your amazon ec2 resources. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html, 2018.
- [BBC⁺94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [Boe14] Carl Boettiger. An introduction to docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014. URL: <http://arxiv.org/abs/1410.0846>, [arXiv:1410.0846](https://arxiv.org/abs/1410.0846).
- [Cur18] Chad Curtis. `diff_classifier`. https://github.com/ccurtis7/diff_classifier, 2018.
- [GBA⁺14] Eleftherios Garyfallidis, Matthew Brett, Bagrat Amirbekian, Ariel Rokem, Stefan Van Der Walt, Maxime Descoteaux, and Ian Nimmo-Smith. Dipy, a library for the analysis of diffusion mri data. *Frontiers in Neuroinformatics*, 8:8, 2014. doi:10.3389/fninf.2014.00008.

6. The "best" configuration could be specified by the user on Knot instantiation as either the one which minimizes cost to the user or that which minimizes the wall time required to process the input data.

- [GSM⁺16] Matthew F Glasser, Stephen M Smith, Daniel S Marcus, Jesper L R Andersson, Edward J Auerbach, Timothy E J Behrens, Timothy S Coalson, Michael P Harms, Mark Jenkinson, Steen Moeller, Emma C Robinson, Stamatios N Sotiropoulos, Junqian Xu, Essa Yacoub, Kamil Ugurbil, and David C Van Essen. The human connectome project's neuroimaging approach. *Nat. Neurosci.*, 19(9):1175–1187, August 2016.
- [HTdAC⁺14] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspoul Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suci. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. ACM.
- [JPV⁺17] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. *ArXiv e-prints*, February 2017. [arXiv:1702.04024](https://arxiv.org/abs/1702.04024).
- [Kra17] Vadim Kravcenko. pipreqs. <https://github.com/bndr/pipreqs>, 2017.
- [MDZ⁺17] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSaiyad. Comparative evaluation of big-data systems on scientific image analytics workloads. *Proceedings of the VLDB Endowment*, 10(11):1226–1237, 2017.
- [Nan17] Elizabeth Nance. Brain-Penetrating nanoparticles for analysis of the brain microenvironment. *Methods Mol. Biol.*, 1570:91–104, 2017.
- [NEZH⁺17] Anwar O Nunez-Elizalde, Tianjiao Zhang, Alexander G Huth, James S Gao, Storm Slivkoff, Mark D Lescroart, Fatma Deniz, Carson McNeil, Robert Gibboni, Sara F Popham, Ariel Rokem, Michael D Oliver, and Jack L Gallant. cottoncandy: scientific python package for easy cloud storage, October 2017. URL: <https://doi.org/10.5281/zenodo.1034342>, doi:10.5281/zenodo.1034342.
- [NWS⁺12] Elizabeth A Nance, Graeme F Woodworth, Kurt A Sailor, Ting-Yu Shih, Qingguo Xu, Ganesh Swaminathan, Dennis Xiang, Charles Eberhart, and Justin Hanes. A dense poly (ethylene glycol) coating improves penetration of large polymeric nanoparticles within brain tissue. *Sci. Transl. Med.*, 4(149):149ra119–149ra119, 2012.
- [RHR18a] Adam Richie-Halford and Ariel Rokem. Cloudknot documentation. <https://richford.github.io/cloudknot/index.html>, 2018.
- [RHR18b] Adam Richie-Halford and Ariel Rokem. Cloudknot examples. <https://github.com/richford/cloudknot/tree/master/examples>, 2018.
- [RHR18c] Adam Richie-Halford and Ariel Rokem. Cloudknot repository. <https://github.com/richford/cloudknot>, 2018.
- [Roc15] M Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference (Scipy 2015)*, 2015.
- [Rod17] Daniel Rodriguez. dask-ec2 repository. <https://github.com/dask/dask-ec2>, 2017.
- [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, February 2017.
- [Wan16] Brian A Wandell. Clarifying human white matter. *Annu. Rev. Neurosci.*, April 2016.
- [ZCF⁺10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10. static.usenix.org, 2010.

Equity, Scalability, and Sustainability of Data Science Infrastructure

Anthony Suen^{§*}, Laura Norén[‡], Alan Liang[§], Andrea Tu[§]



Abstract—We seek to understand the current state of equity, scalability, and sustainability of data science education infrastructure in both the U.S. and Canada. Our analysis of the technological, funding, and organizational structure of four types of institutions shows an increasing divergence in the ability of universities across the United States to provide students with accessible data science education infrastructure, primarily JupyterHub. We observe that generally liberal arts colleges, community colleges, and other institutions with limited IT staff and experience have greater difficulty setting up and maintaining JupyterHub, compared to well-funded private institutions or large public research universities with a deep technical bench of IT staff. However, by leveraging existing public-private partnerships and the experience of Canada’s national JupyterHub (Syzygy), the U.S. has an opportunity to provide a wider range of institutions and students access to JupyterHub.

Index Terms—data science education, Jupyter, Jupyterhub, higher education

Introduction

Data science education has experienced great demand over the past five years, with increasing numbers of programs and majors being developed. This demand has fueled the growth of JupyterHubs, which create on-demand, cloud based Jupyter notebooks for students and researchers. Compared to local environments that run Jupyter, a cloud based JupyterHub provides many conveniences including not requiring any installation, quicker access to course content, and computing flexibility, such that users even on Chromebooks or iPads are able to run Jupyter notebooks.

Additional benefits include the ability to quickly deploy customizations for different use cases, authentication, autograding, and providing campus-wide computing and storage. Overall, universities have found that utilizing JupyterHubs increases accessibility to data science tools, improves the scaling of data science and computing courses into many other domains, and provides a cohesive learning and research platform.

However, little was known about the barriers universities face when attempting to deploy JupyterHub, which has only been in use since 2015.

This paper aims to understand how JupyterHub is affecting the equity, scalability, and sustainability of data science education by providing four cases studies of how JupyterHubs are being

deployed in varying academic institutions across the United States and Canada. We will look at the barriers to deploy, maintain, and grow JupyterHub from the technical staffing and financial perspectives of each institution. The four case studies include large and technical universities such as UC Berkeley, small liberal arts colleges, private universities with large endowments like Harvard, and the Canadian National JupyterHub Model.

We conducted over 10 qualitative interviews with university faculty and IT staff from around the U.S. and Canada. We also reviewed documentation found on Github and websites of 20 institutions regarding their JupyterHub deployments. We structured our analysis by first trying to understand the institution’s educational goals and how it drives funding and decision/structure. We then delve into the infrastructural costs, capabilities, along with team size. We lastly measured educational impact, such as the number of students served and the number of classes provided. We conclude with a summary of the findings and potential ways to improve equity, scalability, and sustainability of current existing JupyterHub infrastructure.

Case Study 1: UC Berkeley

In Spring 2015, UC Berkeley became one of the first universities to adopt JupyterHub¹. Initially set up for 100 students in the new Foundations of Data Science Course *Data 8*, the JupyterHub instance has quickly expanded to now support over 1,000 students in *Data 8* each semester and more than 3,000 students in Berkeley’s *Data Science* connectors, modules, and upper division courses. An additional 45,000 students utilize the JupyterHub in *Data 8*’s free online EdX version.

UC Berkeley aims to serve large portions of its 30,000 undergraduates with data science tools, thus creating the motivation for it to build one of the largest JupyterHub deployments in the world. This cross campus pedagogical vision is assisted by the presence of a large technical team, which consists of many members of the core Jupyter team. UC Berkeley’s JupyterHub runs on the Kubernetes platform, which allows for easily scalable clusters that can support many thousands of users. Furthermore, Berkeley’s JupyterHub infrastructure, which subsists on cloud credits, is supported by long running industry relations and partnerships with cloud vendors like Microsoft and Google.

The UC Berkeley infrastructure team in charge of running Berkeley’s instance of JupyterHub, known as “Datahub”, consists of the Dean of the Division of Data Sciences, one tenured teaching faculty, one full-time staff member, ~10 postdocs and graduate students who can help troubleshoot—many of which are from

* Corresponding author: anthonymsuen@berkeley.edu

§ University of California, Berkeley

‡ New York University

the core Jupyter team—along with a large, technically proficient undergraduate support staff².

UC Berkeley’s model faces sustainability challenges given its heavy reliance on undergraduates, graduate students and postdoc staff and donated computing credits from cloud vendors. Student and postdoc staff generally move on and have other priorities to advance their careers as they typically do not advance their careers by doing SysAdmin work, leading to a lack of consistent support staff and a consequent lack of consistent expertise. The reliance on free cloud credits is further not guaranteed forever and requires regular negotiations with public cloud vendors.

Nonetheless, Berkeley’s model benefits from its campus-wide scale, setting the ground for a large and diverse array of data science courses to be setup with minimum infrastructure overhead³. The infrastructure can also support very large courses, like quantitative gateway courses for many departments. The Berkeley Datahub has a workflow with unique features like interactive links and Ok.py for large scale autograding of thousands of assignments. Finally, it provides a common suite of tools that are widely accessible, allowing students a productive and cohesive environment for both learning and research.

Case Study 2: Small Liberal Arts Universities

The team interviewed several small liberal arts colleges to see how they utilized Jupyter in their data science or computer science curricula. We learned that lack of funding, insufficient technical knowledge, limited relationships and experiences dealing with cloud vendors, and a shortage of time from busy instructors seem to be the major hurdles to deploying a successfully running JupyterHub.

At liberal arts colleges, deployments are usually designed for small classes consisting of ~20-30 students and maintained by one or two professors. There exists little IT help for the professor, as compared to the vast number of support staff at institutions like UC Berkeley. Some smaller institutions have even asked public institutions like UC Berkeley for support. The lack of proper guidance and departmental resources, along with overburdened faculty, often may dissuade efforts to set up JupyterHub altogether. Generally, paying for such technology is also tough and ad hoc for smaller institutions.

One of the exceptions is Bryn Mawr College; its JupyterHub deployment currently hosts and allows access to a wide range of courses. Some courses such as *Introduction to Computing* (introductory computer science course) have migrated to the JupyterHub environment, while new courses such as *Computing in Biology* have been introduced specifically utilizing Jupyter. Bryn Mawr has emphasized using JupyterHub due to its accessibility for biology students who have limited experienced with programming, while also making it useful for CS students who are interested in biological applications for CS. The *Bio/CS 115: Computing Through Biology* course⁴, which was developed based on the Jupyter environment, serves as an alternative CS intro course and a 2nd semester Biology intro course. This option reduces the prerequisite barriers of entry to both domains and allows students to learn both in a well-integrated manner, especially given the amount of intro courses that compete for their schedules.

Case Study 3: Wealthy Private Universities

Compared to smaller liberal arts universities, well-funded private universities often have a rich suite of IT resources. Even if

internal IT staff encounter limitations, well funded private universities often pay third-party vendors to help deploy and maintain JupyterHubs and all related support infrastructure. Harvard has said that they “hired a firm to help us implement JupyterHub on AWS”. Compared to smaller liberal arts colleges, the experience is relatively free of frustration since the university covers all costs. Nonetheless, Harvard has noted that using JupyterHub has increased flexibility and hence decreased setup costs for both users and instructors, and has further claimed that this solution is much more cost effective compared to traditional solutions.

Most of the classes that have deployed JupyterHub are still relatively small, with most having 12-50 students. At Harvard, JupyterHub was deployed on AWS for two classes in the School of Engineering, which provided significant customization. The Signal Processing class used a Docker-based JupyterHub, where each user was provisioned with a docker container notebook. For the Decision Theory class, JupyterHub used a dedicated EC2 instance per user’s notebook, providing better scalability, reliability and cost efficiency⁵. Harvard’s School of Engineering and Applied Science (SEAS) further announced in October 2017 for a schoolwide JupyterHub deployment⁶. In addition to SEAS’s JupyterHub, the Harvard Medical School has its own JupyterHub deployment.

Instead of deploying and maintaining their own JupyterHubs, other universities have found success by contracting a third-party vendor to deploy JupyterHub. Vocareum⁷, an example of one company specializing in this space, helps to set up and manage environments like Jupyter and hosts labs for students to access. Currently, their data sciences lab is used by many wealthy private universities including Cornell, Columbia, and the University of Notre Dame. Others firms that provide similar services include CoCalc and Gryd.

However, the majority of universities generally have less experience with cloud computing and experienced IT staff, thus limiting the replicability of the model. Furthermore, most universities’ data science initiatives cannot rely on their university’s operating budget to support this type of teaching expense, especially if classes are relatively small (12-50 students), hindering scalability of the model. If done in an uncoordinated way, the costs can skyrocket if departments independently contract with cloud providers and IT consultants to set up their own JupyterHubs.

Case Study 4: Canadian Federation (PIMS)

In 2017, an initiative in Canada led by the Pacific Institute of Mathematics and Sciences (PIMS) and hosted by Compute Canada started a new national model for JupyterHub that provides access to numerous institutions across Canada⁸. With data privacy laws removing the option of using cloud service providers, Syzygy grew to become the largest federally funded JupyterHub and is utilized by more than 8,000 students across 15 universities in Canada. Syzygy is run and supported by one full-time system network manager based at PIMS who oversees installations and collaborates with IT staff at Compute Canada. Any Canadian University can simply ask Syzygy for a JupyterHub and a new cluster will be set up. The system manager is paid for by Compute Canada, and further grants from the Canadian federal government (\$4.5m) and Alberta (\$1m) support professors and teachers. There is also time donation from professors at 10 different institutions.

Syzygy has some potential bottlenecks. Firstly, there is only one dedicated staff member conducting core management and

operations for 15 different institutions. Some scaling issues also currently exist as any institution's JupyterHub is at most able to handle ~2 classes of students concurrently (around 200-300 students). Nonetheless, this is a functional model in terms of scale and sustainability based on the number of universities involved, Canada's population size, and strong governmental support.

The leaders of the effort believe that there are multiple benefits to the strategy. Firstly, it can accommodate small classes, modules, and even high schools across the country. Secondly, it allows instructors to focus more on course development, instead of operating a JupyterHub. Thirdly, it fosters better cross university collaboration by sharing experiences and course modules through a common network.

Conclusion - A Path Forward to a National Jupyterhub

While the grassroots efforts across the U.S. have sparked significant innovation in the realm of data science education infrastructure, it has also created a growing chasm of capabilities between institutions. To equitably increase the access to JupyterHub requires a new model to support many smaller institutions.

Today, only large public or wealthy private universities in the U.S. can provide JupyterHub for many undergraduates. At smaller resource-constrained institutions, deploying a JupyterHub instance for a single class possesses nontrivial costs and may be daunting for one instructor or their university IT staff. Unfortunately, if there is no alternative way to access JupyterHub for data science education, smaller less well-funded institutions and underrepresented communities cannot utilize JupyterHub.

When considering the future of JupyterHub in higher data science education, we see four potential pathways:

- **Status Quo** - Continuing the current grassroots and uncoordinated JupyterHub deployments across institutions would mean smaller or less resource rich institutions would likely continue to face existing barriers. For smaller and resource constrained institutions, JupyterHub would continue to experience very low slow rates of adoption.
- **Institutional Grants** - Increasing foundational or governmental funding for individual universities to set up their JupyterHubs is another option. Funding can enable individual institutions to hire IT staff or pay third-party vendors to create a JupyterHub environment. Based on Berkeley's and Harvard's experiences, we've concluded that grants to hire staff to deploy Jupyterhub is non-scalable given the high costs of hiring IT staff with such specialized experience. Funding third-party vendors like CoCalc, Gryd, Vocareum and public cloud providers like Google or Microsoft to help set up individual JupyterHubs is conceivable, but the individual nature of these transactions may end up being more costly than potential coordinated national or regional models.
- **A National JupyterHub** - A national JupyterHub would offer cost benefits such as utilizing existing federally funded national supercomputing centers. However, a single national hub is difficult to realize due to high coordination costs with thousands of universities.
- **Regional Hubs Model** - Given the number of universities in the U.S., establishing several regional hubs can reduce the burden of deployment and maintenance costs that individual universities experience today. For each regional network, by deploying a large Kubernetes cluster that can

support many thousands of users, individual universities can then deploy their own JupyterHubs on the cluster.

The West Big Data Innovation Hub, UC Berkeley, and Microsoft will be launching a pilot program by setting up a Kubernetes cluster using Azure for a small group of Western U.S. universities to pilot their JupyterHubs starting in the Summer of 2018. This will lower the administrative burden while providing a free scalable infrastructure solution for many small or resource constrained universities. Further integration of regional computing facilities at major research universities should be investigated.

1. Kim, A. (2018, May 2). The Jupyterhub Journey: Starting Small and Scaling Up. Retrieved July 5, 2018, from <https://data.berkeley.edu/news/jupyterhub-journey-starting-small-and-scaling>
2. Suen, A. (2018, March 15). People. Retrieved July 5, 2018, from <https://data.berkeley.edu/about/people>
3. Kim, A. (2018, February 20). Modules: Data Made Accessible to Many. Retrieved July 5, 2018, from <https://data.berkeley.edu/news/modules-data-made-accessible-many>
4. Shapiro, J. (2017, May 20). Computing Through Biology with Jupyter. Speech presented at Jupyter Day Philly, Philadelphia. Retrieved May 24, 2018, from https://github.com/BrynMawrCollege/TIDES/blob/master/JupyterDayPhilly/JAShapiro_JupyterDayPhilly_2017-05-19.pdf
5. Harvard. (2018). cloudJHub. Retrieved May 24, 2018, from <https://github.com/harvard/cloudJHub>
6. Ba, D. (2017, October 23). SEAS Computing and Academic Technology for FAS Launch JupyterHub Canvas Integration. Retrieved July 6, 2018, from <https://atg.fas.harvard.edu/news/seas-computing-and-academic-technology-fas-launch-jupyterhub-canvas-integration>
7. DATA SCIENCES LAB @ VOCAREUM. (n.d.). Retrieved July 6, 2018, from <https://www.vocareum.com/home/data-sciences-lab/>
8. Canadians Land on Jupyter. (2017, July 11). Retrieved May 24, 2018, from <https://www.pims.math.ca/news/canadians-land-jupyter>
9. Mandava, V. (2017, June 8). NSF Big Data Innovation Hubs collaboration - looking back after one year - Microsoft Research. Retrieved May 24, 2018, from <https://www.microsoft.com/en-us/research/blog/nsf-big-data-innovation-hubs-collaboration/>

Composable Multi-Threading and Multi-Processing for Numeric Libraries

Anton Malakhov^{‡*}, David Liu[‡], Anton Gorshkov^{‡†}, Terry Wilmarth[‡]

<https://youtu.be/HKjM3peINTw>

Abstract—Python is popular among scientific communities that value its simplicity and power, especially as it comes along with numeric libraries such as [NumPy], [SciPy], [Dask], and [Numba]. As CPU core counts keep increasing, these modules can make use of many cores via multi-threading for efficient multi-core parallelism. However, threads can interfere with each other leading to overhead and inefficiency if used together in a single application on machines with a large number of cores. This performance loss can be prevented if all multi-threaded modules are coordinated. This paper continues the work started in [AMala16] by introducing more approaches to coordination for both multi-threading and multi-processing cases. In particular, we investigate the use of static settings, limiting the number of simultaneously active [OpenMP] parallel regions, and optional parallelism with Intel® Threading Building Blocks (Intel® [TBB]). We will show how these approaches help to unlock additional performance for numeric applications on multi-core systems.

Index Terms—Multi-threading, Multi-processing, Oversubscription, Parallel Computations, Nested Parallelism, Multi-core, Python, GIL, Dask, Joblib, NumPy, SciPy, TBB, OpenMP

1. Motivation

A fundamental shift toward parallelism was declared more than 11 years ago [HSutter], and today, multi-core processors have become ubiquitous [WTichy]. However, the adoption of multi-core parallelism in the software world has been slow and Python along with its computing ecosystem is not an exception. Python suffers from several issues which make it suboptimal for parallel processing.

In particular, Python’s infamous global interpreter lock [GIL] makes it challenging to scale an interpreter-dependent code using multiple threads, effectively serializing them. Thus, the practice of using multiple isolated processes is popular and widely utilized in Python since it avoids the issues with the GIL, but it is prone to inefficiency due to memory-related overhead. However, when it comes to numeric computations with libraries like Numpy, most of the time is spent in C extensions with no access to Python data structures. The GIL can be released during such computations, which enables better scaling of compute-intensive applications. Thus, both multi-processing and multi-threading approaches are valuable for Python users and have their own areas of applicability.

* Corresponding author: Anton.Malakhov@intel.com

‡ Intel Corporation

† These authors contributed equally.

Copyright © 2018 Anton Malakhov et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Scaling parallel programs is challenging. There are two fundamental laws which mathematically describe and predict scalability of a program: Amdahl’s Law and Gustafson-Barsis’ Law [AGlaws]. According to Amdahl’s Law, speedup is limited by the serial portion of the work, which effectively puts a limit on scalability of parallel processing for a fixed-size job. Python is especially vulnerable to this because it makes the serial part of the same code much slower compared to implementations in other languages due to its deeply dynamic and interpretative nature. In addition, the GIL serializes operations that could be potentially executed in parallel, further adding to the serial portion of a program.

Gustafson-Barsis’ law states that if the problem size grows along with the number of parallel processors, while the serial portion grows slowly or remains fixed, speedup increases as processors are added. This law eases the concerns regarding Python as a language for parallel computing when the amount of serial computation in a Python code is fixed, and all the data-processing is hidden behind libraries like NumPy and SciPy. However, a larger problem size demands more operational memory to compute, but memory is a limited resource. Even if problem size is nearly unlimited, as it is for "Big Data", it still has to be decomposed into chunks that fit into memory. This limited growth of the problem size on a single node results in the scalability limitations defined by Amdahl’s Law anyway. Thus, the best strategy to efficiently load a multi-core system is still to avoid serial regions and synchronization.

1.1. Nested Parallelism

To avoid serial regions, we expose parallelism at all possible levels of an application. For example, we make outermost loops parallel, and explore functional, flow graph, or pipeline types of parallelism on the application level. Python libraries that help to achieve this are Dask, Joblib, and the built-in `multiprocessing` and `concurrent.futures` modules. On the innermost level, data-parallelism can be delivered by Python modules like [NumPy] and [SciPy]. These modules can be accelerated with optimized math libraries like Intel® Math Kernel Library (Intel® [MKL]), which is multi-threaded internally using OpenMP (with default settings).

When everything is combined together, the situation arises where code from one parallel region calls a function with another parallel region inside. This is called *nested parallelism*.

1.2. Issues of Oversubscription

The libraries named above do not coordinate the creation or pooling of threads, which may lead to *oversubscription*, a situation in which there are many more active software threads than available hardware resources. For sufficiently big machines with roughly more than 16 cores, oversubscription can lead to sub-optimal execution due to frequent context switches, excessive thread migration, poor cache locality, and load imbalance.

For example, Intel OpenMP* runtime library (used by NumPy/SciPy) may keep its threads active to facilitate the rapid start of subsequent parallel regions. This is usually a useful approach to reduce work distribution overhead, but when another active thread pool exists in the application, it can impact performance. This is because the waiting OpenMP worker threads consume CPU time busy-waiting, while the other parallel work cannot start until OpenMP threads stop spinning or are preempted by the OS.

Because overhead from linear oversubscription (e.g. 2x) is not always visible on the application level (especially for smaller numbers of processor cores), it can be tolerated in many cases when the work for parallel regions is big enough to hide the overhead. However, in the worst case, a program starts multiple parallel tasks and each of these tasks ends up executing an OpenMP parallel region. This results in quadratic oversubscription (with default settings) which ruins multi-threaded performance on systems with a significant number of threads. For some larger systems like Intel® Xeon Phi™, it may not even be possible to create as many software threads as the number of hardware threads squared due to insufficient resources.

1.3. Threading Composability

The co-existing issues of multi-threaded components together define the *threading composability* of a program module or component. A perfectly composable component should be able to function efficiently among other such components without affecting their efficiency. The first aspect of building a composable threading system is to avoid creation of an excessive number of software threads, preventing oversubscription. Ideally, a component or a parallel region should not dictate how many threads it needs for execution (*mandatory parallelism*). Instead, components or parallel regions essentially expose available parallelism to a runtime library, which in turn can provide control over the number of threads or can automatically coordinate tasks between components and parallel regions and map them onto available software threads (*optional parallelism*).

1.4. Restricting Number of Threads used in Nested Levels

A common way to solve oversubscription issues involving the OpenMP runtime library is to disable nested parallelism or to carefully adjust it according to the number of application threads. This is usually accomplished by setting environment variables controlling the OpenMP runtime library. For example, `OMP_NUM_THREADS=1` restricts the number of threads used in an OpenMP parallel region to 1. We do not discourage the use of this approach as it might be sufficient to solve the problem for many use cases. However, this approach can have potential performance-reducing drawbacks:

- 1) There may not be enough parallelism at the outer application level. Blindly disabling nested parallelism can result in underutilization, and consequently, slower execution.
- 2) Globally setting the number of threads once does not take into account different components or phases of the application, which can have differing requirements for optimal performance.
- 3) Setting the optimal value requires the user to have a deep understanding of the issues, the architecture of the application, and the system it uses.
- 4) There are additional settings to take into account like `KMP_BLOCKTIME` (time a thread spins before going to sleep) and thread affinity settings.
- 5) The issue is not limited to OpenMP. Many Python packages like Numba, PyDAAL, OpenCV, and Intel's optimized SciKit-Learn are based on Intel® TBB or a custom threading runtime.

2. New approaches

Our goal is to provide alternative solutions for composing multiple levels of parallelism across multiple threading libraries with same or better performance compared to the usual approaches. At the same time, we wish to keep the interface for this simple, requiring shallower knowledge and fewer decisions from end-users. We evaluate several new approaches in this paper.

2.1. Static Settings

A common way to parallelize Python code is to employ process or threads *pools* (or *executors*) provided through a standard library. These pools are also used by other Python libraries implementing parallel computations like Dask and Joblib. We modify these pools so that each pool worker calling a nested parallel computation can only use a particular number of processor cores.

For example, if we have an eight core CPU and want to create a pool of two workers, we limit the number of threads per pool worker to four. When using a process pool, we set the thread affinity mask for each worker process so that any threads created within a particular process operate only on a specific set of processor cores. In our example, the first process will use cores 0 through 3 and the second process will use cores 4 through 7. Since both OpenMP and Intel® TBB respect the incoming affinity mask during initialization, they limit the number of threads per process to four. As a result, we have a simple way of sharing threads between pool workers without any oversubscription issues.

When a multi-threading pool is used for application-level parallelism, the idea is the similar. Instead of setting process affinity masks, we limit the number of threads per pool worker using the threading runtime API. For example, we can use `omp_set_num_threads()` to limit the number of threads for OpenMP parallel regions. This approach is similar to how `OMP_NUM_THREADS` environment variable can be specified for the entire application. The difference is that here, we can use knowledge of how many outermost workers are requested by the application and how much hardware parallelism is available on the machine, and then calculate an appropriate number of threads automatically and apply it for the specific pool instance. This is a more flexible approach for applications which might use pools of different sizes within the same run.

To implement this approach, we have created a Python module called *smp* (static or symmetric multi-processing). It works with

*. Other names and brands may be claimed as the property of others.

both thread and process pools from `multiprocessing` and `concurrent.futures` modules using the *monkey patching* technique that enables us to use this solution without any code modifications in user applications. To run it, we use one of the following commands:

```
python -m smp app.py
python -m smp -f <oversubscription_factor> app.py
```

The optional argument `-f <oversubscription_factor>` sets an oversubscription factor that will be used to compute the number of threads per pool worker. By default it is 2, which means that in our example, 8 threads will be used per process. By allowing this limited degree of oversubscription by default, many applications achieve better load balance and performance that will outweigh the overhead incurred by the oversubscription, as discussed in section 3.5. For the particular examples we show in this paper, the best performance is achieved with an oversubscription factor of 1 specified on the command line as `-f 1`, indicating that any amount of oversubscription leads to non-optimal performance for those applications.

2.2. Limiting Simultaneous OpenMP Parallel Regions

The second approach relies on extensions implemented in the Intel® OpenMP runtime. The basic idea is to prevent oversubscription by not allowing multiple parallel regions (on different top-level application threads) to run simultaneously. This resembles the "Global OpenMP Lock" that was suggested in [AMala16]. The implementation provides two modes for scheduling parallel regions: *exclusive* and *counting*. Exclusive mode implements an exclusive lock that is acquired before running a parallel region and released after the parallel region completes. Counting mode implements a mechanism equivalent to a semaphore, which allows multiple parallel regions with small number of threads to run simultaneously, as long as the total number of threads does not exceed a limit. When the limit is exceeded, the mechanism blocks in a similar way to the exclusive lock until the requested resources become available. This idea is easily extended to the multiple process case using Inter-Process Coordination (IPC) mechanisms such as a system-wide semaphore.

The exclusive mode approach is implemented in the Intel® OpenMP* runtime library being released as part of Intel® Distribution for Python 2018¹ as an experimental preview feature, later the counting mode was also added. Setting the `KMP_COMPOSABILITY` environment variable as follows should enable each OpenMP parallel region to run exclusively, eliminating the worst oversubscription effects:

```
env KMP_COMPOSABILITY=mode=exclusive python app.py
env KMP_COMPOSABILITY=mode=counting python app.py
```

With composability mode in use, multi-processing coordination is enabled automatically on the first usage. Each process has its own pool of OpenMP worker threads. While these threads are coordinated across the processes preventing oversubscription, creating a large number of threads per process can still cause resource exhaustion.

2.3. Coordinated Thread Pools with Intel® TBB

Our last approach was introduced in a previous paper [AMala16]. It is based on using Intel® TBB as a single engine for coordinating

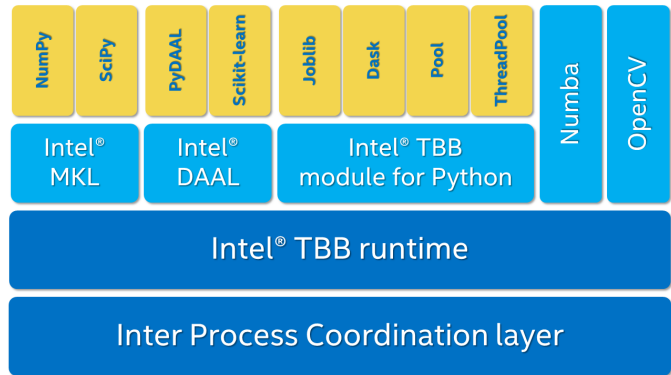


Fig. 1: Intel® TBB provides a common runtime for Python modules and coordinates threads across processes.

parallelism across all Python pools and modules. TBB's work stealing task scheduler is used to map tasks onto a limited set of TBB worker threads while the monkey-patching technique is applied in a TBB module for Python that implements Python's `ThreadPool` on top of TBB tasks. This approach makes it possible to dynamically balance the load across multiple tasks from different modules but is limited to the multi-threading case.

In this paper, we extended this approach by introducing an InterProcess Communication (IPC) layer for Intel® TBB. As shown in figure 1, different modules that are combined into a single application, work on top of the shared Intel® TBB pool, which is coordinated across multiple processes.

The TBB module for Python introduces a shared library, *libirml*, which is recognized by Intel® TBB library as a thread pool provider. Before creating any new worker thread, this library acquires an IPC semaphore. The semaphore is initialized with maximum value set to the number of CPU hardware threads. When all the allowed threads are allocated, no additional threads can be created.

Because of this greedy algorithm, some TBB processes can be left without worker threads at all. This is a legitimate situation within the optional parallelism paradigm implemented in Intel® TBB, which does not prevent master threads from making progress and completing computation even without worker threads joined. Thus, even in the worst case, counting all the worker and master threads, the total number of active threads for all the running processes does not exceed twice the number of CPU hardware threads.

When the first process finishes its computation, TBB puts the worker threads back in the pool and releases resources for the semaphore. A special monitor thread implemented in *libirml* detects this situation and the rest of the processes are allowed to acquire the relinquished resources and to add threads on the fly to ongoing computations in order to improve CPU utilization.

However, if we don't remove excess threads, this solution does not prevent resource exhaustion. Since we cannot move threads from one process to another, there can be too many threads allocated at the same time. This prevents processes with fewer threads from creating more threads to balance the load. To fix this issue, we implemented an algorithm that disposes of unused threads when a shortage of resources is detected.

This TBB-based approach to coordination is more dynamic and flexible than one based on OpenMP because it allows to re-purpose and rebalance threads more flexibly, achieving better load

¹ It was also introduced on Anaconda cloud starting with the version 2017.0.3 in limited, undocumented form.

balancing overall. Even in counting composability mode, OpenMP needs to wait for all the requested threads to become available, while Intel® TBB allows threads to join parallel computations already in progress.

The TBB IPC module should be enabled manually via explicit command line key `--ipc`, for example:

```
python -m tbb --ipc app.py
```

3. Evaluation

The results for this paper were acquired on a 2-socket system with Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz (22 cores * 2 hyper-threads) and 256GB DDR4 @ 2400 MHz. This system consists of 88 hardware threads in total.

For our experiments, we used [Miniconda] distribution along with the packages of Intel® Distribution for Python [IntelPy] installed from anaconda.org/intel

```
# activate miniconda
source <path to miniconda3>/bin/activate.sh
# create & activate environment from the Intel channel
conda create -n intel3 -c intel numpy dask tbb4py smp
source activate.sh intel3
# this setting is used for default runs
export KMP_BLOCKTIME=0
```

We installed the following versions and builds of the packages for our experiments: Python 3.6.3-intel_12, numpy 1.14.3-py36-intel_0, dask 0.18.1-py36_0, mkl 2018.0.3-intel_1, openmp 2018.0.3-intel_0, tbb4py 2018.0.4-py36_0, smp 0.1.3-py_2.

Here is an example of how to run the benchmark programs in different modes:

```
# Default mode (with KMP_BLOCKTIME=0 in effect)
python bench.py
# Serialized OpenMP mode
env OMP_NUM_THREADS=1 python bench.py
# SMP module, oversubscription factor = 1
python -m smp -f 1 bench.py
# Composible OpenMP, exclusive mode
env KMP_COMPOSABILITY=mode=exclusive python bench.py
# Composible OpenMP, counting mode
env KMP_COMPOSABILITY=mode=counting python bench.py
# Composible TBB mode (multithreading only)
python -m tbb bench.py
# Composible TBB mode with IPC on
python -m tbb --ipc bench.py
```

For our examples, we will talk mostly about the multi-threading case, but according to our investigations, all conclusions that will be shown are applicable for the multi-processing case as well unless additional memory copying happens between the processes, which is out of scope for this paper.

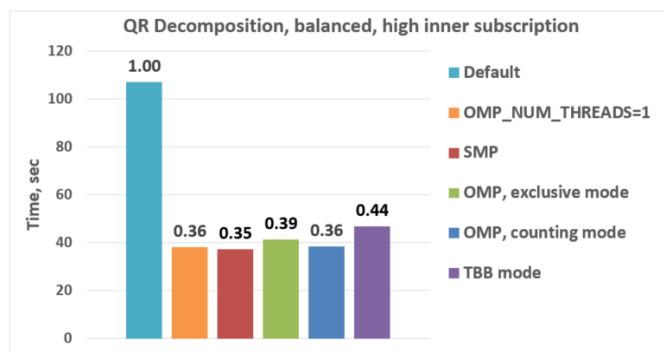


Fig. 2: Execution times for balanced QR decomposition workload.

Please find these benchmarks along with install and run script at [compbench]

3.1. Balanced QR Decomposition with Dask

The code below is a simple program using Dask that validates a QR decomposition function by multiplying computed components and comparing the result against the original input.

```
1 import time, dask, dask.array as da
2 x = da.random.random((440000, 1000),
3                       chunks=(10000, 1000))
4 for i in range(3):
5     t0 = time.time()
6     q, r = da.linalg.qr(x)
7     test = da.all(da.isclose(x, q.dot(r)))
8     test.compute()
9     print(time.time() - t0)
```

Dask splits the array into 44 chunks and processes them in parallel using multiple threads. However, each Dask task executes the same NumPy matrix operations which are accelerated using Intel® MKL under the hood and thus multi-threaded by default. This combination results in nested parallelism, i.e. when one parallel component calls another component, which is also threaded. The execution is repeated numerous times, with results taken from later iterations, in order to avoid the cache-warming effects present in the first iterations.

Figure 2 shows the performance for the code above. By default, Dask processes a chunk in a separate thread, so there are 44 threads at the top level. By default, Dask creates a thread pool with 88 workers, but only half of them are used since there are only 44 chunks. Chunks are computed in parallel with 44 OpenMP workers each. Thus, there can be 1936 threads competing for 44 cores, which results in oversubscription and poor performance.

A simple way to improve performance is to tune the OpenMP runtime using the environment variables. First, we limit the total number of threads. Since we have an 88-thread machine, we limit OpenMP to a single thread per parallel region ((88 CPU threads / 88 workers in thread pool) * 1x over-subscription). We also noticed that reducing the period of time after which an Intel OpenMP worker thread goes to sleep helps to improve performance in workloads with oversubscription (this works best for the multi-processing case but helps for multi-threading as well). We achieve this by setting `KMP_BLOCKTIME` to zero by default. These simple optimizations reduce the computational time by 2.5x.

The third approach using `smp` module and specifying an oversubscription factor of 1 (`-f 1`) does similar optimizations automatically, and shows the same level of performance as for `OMP_NUM_THREADS=1`. The approach is more flexible and works with several thread/process pools in the application scope, even if they have different sizes. Thus, it is a better alternative to manual OpenMP tuning.

The remaining approaches are our dynamic OpenMP- and Intel® TBB-based approaches. Both approaches improve the default result, but OpenMP gives us the fastest time. As described above, the OpenMP-based solution allows processing of chunks one by one without any oversubscription, since each separate chunk can utilize the whole CPU. In contrast, the work stealing task scheduler of Intel® TBB is truly dynamic and uses a single thread pool to process all the given tasks simultaneously. As a

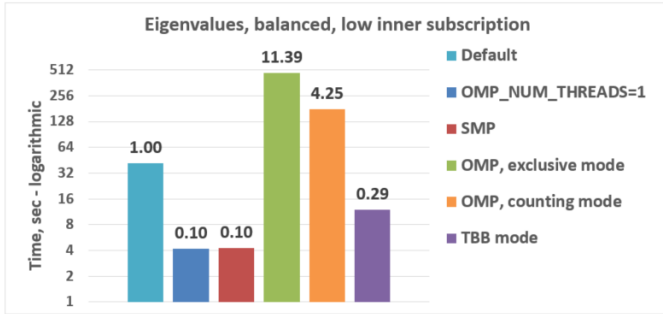


Fig. 3: Execution time for balanced eigenvalues search workload.

result, besides higher overhead for work distribution, it has worse cache utilization.

3.2. Balanced Eigenvalues Search with NumPy

The code below processes eigenvalues and right eigenvectors search in a square matrix using NumPy:

```

1 import time, numpy as np
2 from multiprocessing.pool import ThreadPool
3 x = np.random.random((256, 256))
4 p = ThreadPool(88)
5 for j in range(3):
6     t0 = time.time()
7     p.map(np.linalg.eig, [x for i in range(1024)])
8     print(time.time() - t0)

```

In this example we process several matrices from an array in parallel using Python's ThreadPool while each separate matrix is computed in parallel by Intel® MKL. Similar to the QR decomposition benchmark above, we used quadratic oversubscription here. This code has the distinctive feature that, in spite of parallel execution of eigenvalues search algorithm, it cannot fully utilize all available CPU cores. The additional level of parallelism we use here significantly improves the overall benchmark performance.

Figure 3 shows benchmark execution time using the same modes as in the QR decomposition example. The best choice for this benchmark was to limit number of threads statically either using manual settings or the *smp* module, and obtained about 10x speed-up. Also, Intel® TBB based approach performed much better than composable OpenMP. The reason for this was that there was insufficient parallelism present in each separate chunk. In fact, exclusive compositability mode in OpenMP leads to serial matrix processing, so a significant part of the CPU stays unused. As a result, the execution time in this case becomes even larger than by default. The result of counting mode can be further improved on Intel® MKL side if parallel regions can be adjusted to request fewer threads.

3.3. Unbalanced QR Decomposition with Dask

In previous sections, we discussed balanced workloads where the amount of work per thread at the top level is mostly the same. As we expected, the best strategy for such cases is based on static approaches. However, what if we need to deal with dynamic workloads where the amount of work per thread or process varies? To investigate such cases we have prepared unbalanced versions of our static benchmarks. Each benchmark creates an outermost

2. For more complete information about compiler optimizations, see our Optimization Notice [OptNote]

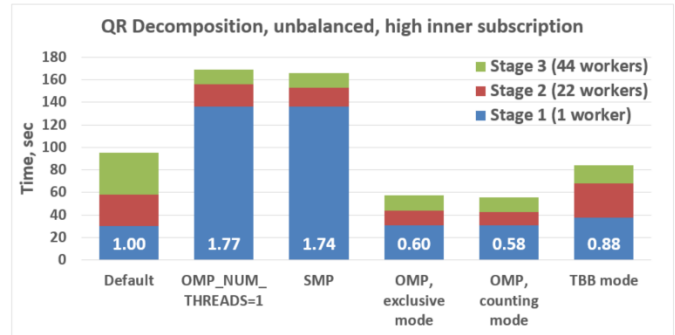


Fig. 4: Execution times for unbalanced QR decomposition workload.

thread pool for 44 workers. We will perform computations in three stages. The first stage uses only one thread from the pool, which is able to fully utilize the whole CPU. During the second stage, half of the top level threads are used (22 in our example). In the third stage, the whole pool is employed (44 threads).

The code below shows this *unbalanced* version of QR decomposition workload:

```

1 import time, dask, dask.array as da
2 def qr(x):
3     t0 = time.time()
4     q, r = da.linalg.qr(x)
5     test = da.all(da.isclose(x, q.dot(r)))
6     test.compute(num_workers=44)
7     print(time.time() - t0)
8 sz = (440000, 1000)
9 x01 = da.random.random(sz, chunks=(440000, 1000))
10 x22 = da.random.random(sz, chunks=(20000, 1000))
11 x44 = da.random.random(sz, chunks=(10000, 1000))
12 qr(x01); qr(x22); qr(x44)

```

Figure 4 demonstrates execution time for all the approaches. The first observation here is that the static SMP approach does not achieve good performance with imbalanced workloads. Since we have a single thread pool with a fixed number of workers, it is unknown which of workers are used and how intensively. Accordingly, it is difficult to set an appropriate number of threads statically. Thus, we limit the number of threads per parallel region based on the size of the pool only. As a result, just a few threads are used in the first stage, which leads to underutilization and slow performance. The second and third stages work well, but overall we have a mediocre result.

The work stealing scheduler of Intel® TBB works slightly better than the default version, but due to redundant work balancing in this particular case it has significant overhead.

The best execution time comes from using composable OpenMP. Since there is sufficient work to do in each parallel region, allowing each chunk to be calculated one after the other avoids oversubscription and results in the best performance.

3.4. Unbalanced Eigenvalues Search with NumPy

The second dynamic example present here is based on eigenvalues search algorithm from NumPy:

```

1 import time, numpy as np
2 from multiprocessing.pool import ThreadPool
3 from functools import partial
4
5 x = np.random.random((256, 256))
6 y = np.random.random((8192, 8192))
7 p = ThreadPool(44)
8

```

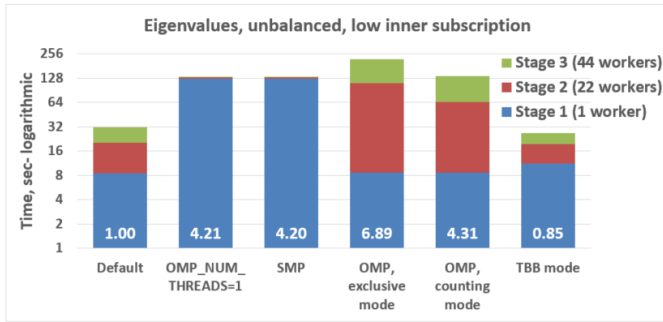


Fig. 5: Execution time for unbalanced eigenvalues search workload.

```

9 t0 = time.time()
10 mmul = partial(np.matmul, y)
11 p.map(mmul, [y for i in range(6)], 6)
12 print(time.time() - t0)
13
14 t0 = time.time()
15 p.map(np.linalg.eig, [x for i in range(1408)], 64)
16 print(time.time() - t0)
17
18 t0 = time.time()
19 p.map(np.linalg.eig, [x for i in range(1408)], 32)
20 print(time.time() - t0)

```

In this workload, we have the same three stages. The second and the third stage computes eigenvalues and the first one performs matrix multiplication. The reason we do not use eigenvalues search for the first stage as well is that it cannot fully load the CPU as we intended.

From figure 5 we can see that the best solution for this workload is Intel® TBB mode, which reduces execution time to 85% of the default mode. SMP module works even slower than the default version due to the same issues as described for the unbalanced QR decomposition example. Composable OpenMP works slower as well since there is not enough work for each parallel region, which leads to CPU underutilization.

3.5. Impact of nested parallelism and oversubscription

The experiments in this section demonstrate the benefits of using nested parallelism and determine what degree of oversubscription impacts performance. We took our balanced eigenvalues search workload (section 3.2) and ran it in default and the best performing SMP modes. Then we ran it with various sizes for the top level thread and process pool, from 1 to 88 workers.

3. For more complete information about compiler optimizations, see our Optimization Notice [OptNote]

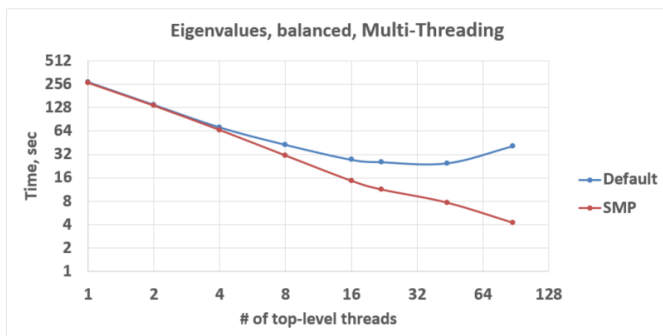


Fig. 6: Multi-threading scalability of eigenvalues search workload.

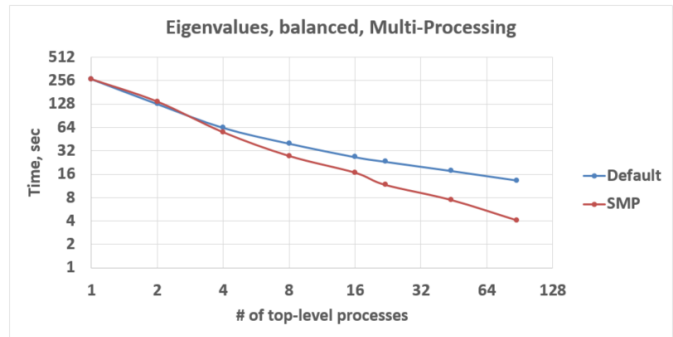


Fig. 7: Multi-processing scalability of eigenvalues search workload.

Figure 6 shows the scalability results for the multi-threading case. The difference in execution time between these two methods starts from 8 threads in top level pool and becomes larger as the pool size increases.

The multi-processing scalability results are shown in figure 7. Multi-processing version differs from multi-threading one by using `multiprocessing.Pool` and returning no result out of the mapped function in order to exclude copying from the measurements. The results are very similar to the multi-threading case: oversubscription effects become visible starting from 8 processes at the top level of parallelization.

4. Solutions Applicability and Future Work

In summary, all three evaluated approaches to compose parallelism are valuable and can provide significant performance increases for both multi-threading and multi-processing cases. Ideally, we would like to find a single solution, which works well in all cases. Instead, the presented approaches complement each other and have their own fields of applicability.

The SMP approach works perfectly for balanced workloads where all the outermost workers have same amount of work. Compared with manual tuning of OpenMP settings, this approach is more stable, since it can work with pools of different sizes within the scope of a single application without performance degradation. Thanks to configuring process affinity mask, it also covers other threading libraries such as Intel® TBB.

The composable OpenMP mode works best with unbalanced benchmarks for cases where there is enough work to load each innermost parallel region.

The dynamic task scheduler from Intel® TBB provides the best performance when innermost parallel regions cannot fully utilize the whole CPU and/or have varying amounts of work to process.

The evidence presented in this paper does not explore the full problem parameter space, however it does provide practical guidance that can be used as a starting point to tune the performance of applications with nested parallelism.

Innermost Parallelism Level	Outermost Parallelism Level		Unbalanced work
	Balanced work		
	Low subscription	High subscription	
Low subscription	\$ python	\$ python -m smp	\$ python -m tbb
High subscription			KMP_COMPOSABILITY

Threads created for blocking I/O operations are not subject to performance degradation caused by oversubscription. In fact, it is recommended to maintain a higher number of threads because

they are mostly blocked in the operating system. If your program uses blocking I/O, please consider using asynchronous I/O instead that blocks only one thread for the event loop and so prevents other threads from being blocked.

We encourage readers to try suggested composability modes and use them in production environments, if this provides better results. However, there are potential enhancements that can be implemented and we need feedback and real-life use cases in order to prioritize the improvements.

Both *tbb* and *smp* modules are implemented and tested with both major Python versions, 2 (starting with 2.7+) and 3 (3.5 and newer). The *smp* module works only on Linux currently, but can be extended to other platforms as well. The *smp* bases calculations only on the size of the pool and does not take into account its real usage. We think it can be improved in future to trace task scheduling pool events and become more flexible.

The composability mode of Intel OpenMP* runtime library is currently limited to Linux platform as well. It works well with parallel regions with high CPU utilization, but it has a significant performance gap in other cases, which we believe can be improved.

The IPC mode of the TBB module for Python is also limited to Linux and classified a preview feature, which might be insufficiently optimized and verified with different use cases. However, the default mode of the TBB module for Python works as well on Windows and Mac OS for multi-threading coordination in single process. Also, the TBB-based threading layer of Intel® MKL might be suboptimal compared to the default OpenMP-based threading layer.

All these problems can be eliminated as more users become interested in using nested parallelism in a production environment and as all software mentioned here is further developed.

5. Conclusion

This paper provides a working definition for threading composability, specifically discussing the necessity for broader usage of nested parallelism on multi-core systems. We also addressed performance issues related to the GIL and oversubscription of threads, for python libraries using parallelism with multi-core processors, such as NumPy, SciPy, SciKit-learn, Dask, and Numba.

Three approaches are suggested as potential solutions. The first approach is to statically limit the number of threads created on the nested parallel level. The second one is to coordinate execution of OpenMP parallel regions. The third one is to use a common threading runtime using Intel® TBB extended to multi-processing parallelism. All these approaches limit the number of active threads in order to prevent penalties of oversubscription. They coordinate parallel execution of independent program modules to improve overall performance.

The examples presented in the paper show promising results while achieving the best performance using nested parallelism in threading composability modes. In particular, balanced QR decomposition and eigenvalues search examples are 2.5x and 7.5x faster compared to the baseline implementations. Imbalanced versions of these benchmarks are 34-35% faster than the baseline.

These improvements are all achieved with different approaches, demonstrating that the three solutions are valuable and complement each other. Our comparison of the suggested approaches provides recommendations for when it makes sense to employ each of them.

All the described modules and libraries are available as open source software and included as part of the free Intel® Distribution for Python product. The Distribution is available as a stand-alone installer [IntelPy] and as a set of packages on anaconda.org/intel channel.

REFERENCES

- [AMala16] Anton Malakhov, "Composable Multi-Threading for Python Libraries", Proc. of the 15th Python in Science Conf. (SCIPY 2016), July 11-17, 2016.
- [NumPy] NumPy, <http://www.numpy.org/>
- [SciPy] SciPy, <https://www.scipy.org/>
- [Dask] Dask, <http://dask.pydata.org/>
- [Numba] Numba, <http://numba.pydata.org/>
- [TBB] Intel(R) TBB open-source site, <https://www.threadingbuildingblocks.org/>
- [OpenMP] The OpenMP(R) API specification for parallel programming, <http://openmp.org/>
- [HSutter] Herb Sutter, "The Free Lunch Is Over", Dr. Dobbs's Journal, 30(3), March 2005. <http://www.gotw.ca/publications/concurrency-ddj.htm>
- [WTichy] Walter Tichy, "The Multicore Transformation", Ubiquity, Volume 2014 Issue May, May 2014. DOI: 10.1145/2618393. <http://ubiquity.acm.org/article.cfm?id=2618393>
- [GIL] David Beazley, "Understanding the Python GIL", PyCON Python Conference, Atlanta, Georgia, 2010. <http://www.dabeaz.com/python/UnderstandingGIL.pdf>
- [AGlaws] Michael McCool, Arch Robison, James Reinders, "Amdahl's Law vs. Gustafson-Barsis' Law", Dr. Dobbs's Parallel, October 22, 2013. <http://www.drdoobs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980>
- [MKL] Intel(R) MKL, <https://software.intel.com/intel-mkl>
- [Joblib] Joblib, <http://pythonhosted.org/joblib/>
- [Miniconda] Miniconda, <https://conda.io/miniconda.html>
- [IntelPy] Intel(R) Distribution for Python, <https://software.intel.com/python-distribution>
- [compbench] Repository for composability benchmarks, https://github.com/IntelPython/composability_bench
- [OptNote] <https://software.intel.com/en-us/articles/optimization-notice>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

The Econ-ARK and HARK: Open Source Tools for Computational Economics

Christopher D. Carroll^{||*}, Alexander M. Kaufman[‡], Jacqueline L. Kazil^{**}, Nathan M. Palmer[§], Matthew N. White[¶]

<https://youtu.be/lytEhrnwu6A>

Abstract—The Economics Algorithmic Repository and toolKit (**Econ-ARK**) aims to become a focal resource for computational economics. Its first ‘framework,’ the Heterogeneous Agent Resources and Toolkit (**HARK**), provides a modern, robust, transparent set of tools to solve a class of macroeconomic models whose usefulness has become increasingly apparent both for economic policy and for research purposes, but whose adoption has been limited because the existing literature derives from idiosyncratic, hand-crafted, and often impenetrable legacy code. We expect future Econ-ARK frameworks (e.g., for analysis of the transmission of beliefs through agents’ social networks) will draw heavily on key elements of the existing HARK framework, including the API, the structure, and documentation standards.

Index Terms—Heterogeneous-Agent Resources toolKit, econ-ark, computational economics, economic modeling

Disclaimer: Views expressed herein do not necessarily reflect the views of the respective institutions that employ the respective authors.

Introduction

The Economics Algorithmic Repository and toolKit (**Econ-ARK**) is a modular programming framework for solving and estimating macroeconomic and macro-financial models in which economic agents can exhibit significant heterogeneity.¹ Models with extensive heterogeneity among agents can be extremely useful for policy and research purposes. However, the most commonly published macroeconomic and macro-finance models have very limited heterogeneity or none at all, in large part because these are the only models that can be easily solved with existing toolkits such as DYNARE [Adjemian2011].

In contrast, models with extensive heterogeneity among agents have no central toolkit and must be solved in a bespoke way. This requires a significant investment of time and human capital before a researcher can produce usable work. This results in needless code duplication, increasing the chance for error and wasting valuable research time. The Econ-ARK project addresses these concerns by providing a set of well-documented code modules that

can be composed together to solve a range of heterogeneous-agent models. Methodological advances in the computational literature allow many types of models to be solved using similar approaches; the Econ-ARK project simply brings these pieces together in one place. HARK is written in Python 2.7, with a pull request underway at the time of this writing to make it fully compatible with both Python 2.7 and 3.6.

Academic research in statistics has standardized on the use of the ‘R’ modeling language for scholarly communication, and on a suite of tools and standards of practice (the use of R-markdown, e.g.) that allow statisticians to communicate their ideas easily to each other. Many other scholarly fields have similarly developed suites of tools that allow scholars to easily and transparently exchange quantitative ideas and computational results without anyone having to master idiosyncratic details of anyone else’s hand-crafted computer code.

The only branch of economics in which anything similar has happened is representative agent (RA) macroeconomics, which (to some degree) has standardized on the use of the DYNARE [Adjemian2011] toolkit for solving representative agent dynamic stochastic general equilibrium models.

We face two primary challenges. The first is to develop a set of resources and examples and standards of practice for communication that are self-evidently a major improvement on the way economists exchange ideas now. The second is to persuade scholars to adopt those tools.

The **Econ-ARK** is the vehicle by which we hope to achieve these objectives. We have begun with the creation of a toolkit for heterogeneous agents (HA) macroeconomics, in part because that is a field where the need for improvement in standards of transparency, openness, and reproducibility is particularly manifested, and partly because it is a field where important progress seems particularly feasible. **QuantEcon** is the most similar project to Econ-ARK and makes use of open source coding tools. However, that project focuses largely on foundational material appropriate for an introductory graduate course on numeric methods in macroeconomics, whereas the Econ-ARK is geared toward the production of new research.²

The traditional approach in macroeconomics has been to assume that aggregate behavior can be understood by modeling the behavior of a single ‘representative agent’ -- the ‘representative consumer’ or ‘representative firm’. HA macroeconomics instead starts by constructing models of the behavior of individual microeconomic agents (a firm or a consumer, e.g.) that match key facts (say, that some people are borrowers and others are

* Corresponding author: ccarroll@jhu.edu

|| Johns Hopkins University

‡ Woodrow Wilson School of Public Policy

** Capital One

§ Econ-ARK

¶ University of Delaware

savers) from the rich microeconomic evidence about the behavior and circumstances of such agents. With that solid foundation in place, macroeconomic outcomes are constructed by aggregating the behavior of the individual agents subject to sensible requirements on the characteristics of the aggregate (such as that the aggregate amount borrowed cannot exceed a function of the aggregate amount saved). For a broad review of representative agent and heterogeneous agents economic modeling, see the discussion by [Guevenen2011] and [Kirman1992]. More broadly, the branch of agent-based macroeconomics explores the issues of emergence and complexity. The interested reader is directed to the Handbooks of Computational Economics, Volumes 2 and 4: [Tseftis2006] and [Hommes2018]. The most recent volume in particular outlines similarities and differences between more traditional heterogeneous agents macroeconomics and so-called "agent-based methods," inspired from fields such as physics and ecology.

The Heterogeneous-Agent Resources toolKit (HARK) is a modular programming framework for solving, estimating, and simulating macroeconomic models with heterogeneous agents. Agents in HARK can be heterogeneous in a large number of ways, such as in wealth, income processes, preferences, or expectations. Models with heterogeneity among agents have proven to be increasingly useful for policy and research purposes.

For example, recent work by [Kaplan2018] has shown that changes in interest rates affect the economy in large part by reallocating income flows across different types of households rather than by causing every household to change their behavior in the same way. The latter implicitly occurs in a traditional rational expectations model, but may be misleading regarding the underlying channel of the effect. [Carroll2017a] shows that the response to fiscal policy (such as stimulus payments or tax cuts) depends crucially on how such payments are distributed across different groups. For example, an extension of unemployment benefits has a bigger effect on spending than a cut in the capital gains tax. [Geanakoplos2010] outlines how heterogeneity drives the leverage cycle, and [Geanakoplos2012] applies these insights to large-scale model of the housing and mortgage markets.

HA models of the kind described above have had a major intellectual impact over the past few years. But the literature remains small, and contributions have come mostly from a few small groups of researchers with close connections to each other. An excellent overview of this literature can be found in the most recent volume of the Handbooks of Computational Economics [Hommes2018] and works cited therein.

In large part, this reflects the formidable technical challenges involved in constructing such models. In each case cited above, the codebase underlying the results is the result of many years of construction of hand-crafted code that has not been meaningfully vetted by researchers outside of the core group of contributors. This is not because researchers have refused to share their code; instead, it is because the codebases are so large, so idiosyncratic, and (in many cases) so poorly documented and organized as to be nearly incomprehensible to anyone but the original authors and their collaborators. Researchers with no connections to the pioneering scholars have therefore faced an unpalatable choice between investing years of their time reinventing the wheel, or investing years of their time deciphering someone else's peculiar and idiosyncratic code.

Researchers who must review the scientific and technical code written by others are keenly aware that the time required

to review and understand another's code can dwarf the time required to simply re-write the code from scratch (conditional on understanding the underlying concepts). This can be particularly important when multiple researchers may need to work on parts of the same codebase, either across time or distance.

The HARK project addresses these concerns by providing a set of well-documented code modules that can be combined to solve a range of heterogeneous-agent models. Methodological advances in the computational economics literature allow many types of models to be solved using similar approaches; the key for HARK is to identify methodologies that are "modular" (in a sense to be described below).

In addition to these methodological advances, the HARK project adopts modern software development practices to ease the burden of code development, code review, code sharing, and collaboration for researchers dealing with computational methods.

Because these problems are generic (and not specific to computational economics), the software development community, and particularly the open-source community, has spent decades developing tools for programmers to quickly consume and understand code written by others, verify that it is correct, and to contribute back to a large and diverse codebase without fear of introducing bugs. The tools used by these professional developers include formal code documentation, unit testing structures, modern versioning systems for automatically tracking changes to code and content, and low-cost systems of communicating ideas, such as interactive programming notebooks that combine formatted mathematics with executable code and descriptive content. These tools operate particularly well in concert with one another, constituting an environment that can greatly accelerate project development for both individuals and collaborative teams. These technical tools are not new-- the HARK project simply aims to apply the best of them to the development of code in computational economics in order to increase researcher productivity, particularly when interacting with other researchers' code.

The rest of this paper will first outline the useful concepts we adopt from software development, with examples of each, and then demonstrate how these concepts are applied in turn to the key solution and estimation methods required to solve heterogeneous-agent models. The sections are organized as follows: Section 1 discusses the natural modular structure of the types of problems HARK solves and provides an overview of the code structure that implements these solutions. Section 2 provides details of the core code modules in HARK. Section 3 outlines two examples that illustrate models in the HARK framework. Section 4 summarizes and concludes.

1. HARK Structure

The class of problems that HARK solves is highly modular by construction. There are approximately these steps in solving a rational heterogeneous agents model:

- 1) Specify the problem faced by an individual agent
- 2) Specify how the actions and states of individual agents collectively generate aggregate outcomes or processes
- 3) For given beliefs about aggregate processes, solve the individual agent's problem
- 4) Simulate the behavior of agents, generating a "history" of aggregate outcomes
- 5) Formulate new beliefs about the aggregate processes based on that history

6) Iterate on steps 3-5 until beliefs converge

In isolation, steps 1 and 3 constitute the solution to a "microeconomic" model in HARK: how an individual agent should optimally act, treating all inputs to his problem as fixed. The inclusion of steps 2, 4, 5, and 6 embeds the microeconomic model in a "macroeconomic" model, requiring consistency among agents' individual behavior, the outcomes that result from the aggregation of these choices, and agents' beliefs about aggregate processes. The assumption of rationality is imposed by having the beliefs formulated in step 5 be justified given the history of aggregate outcomes; agents correctly interpret (a hypothetical) history when forming their new beliefs. Economists call such a solution a "rational expectations equilibrium", as agents' expectations are fulfilled by reality, and they have no reason to update these expectations or beliefs.³

In the section below titled "Sample Model: Perfect Foresight Consumption-Saving," we directly illustrate a microeconomic model in HARK; a full example of a macroeconomic model is outlined in [Carroll2017b].

To *estimate* a model for some research purpose, the economist tries to find the "deep" or "structural" parameters that make model outcomes best match particular features of some dataset. That is, the model is mathematically specified in steps 1 and 2 above, but the economist does not know the values of some vector of model parameters; the objective of the estimation is to find the parameters that make the model best "match" real data. As the dataset, features or moments to match, and particular estimation method (e.g. simulated method of moments or maximum likelihood estimation) are idiosyncratic to each research project, we will not elaborate further here.

In HARK, each of the solution steps is highly modular, and the structure of the solution method suggests a natural division of the code. (The solution method is dynamic programming and fixed point iteration, and the estimation method is Simulated Method of Moments. These are described in detail in [Carroll2012].)

Python modules in HARK can generally be categorized into three types: tools, models, and applications. **Tool modules** contain functions and classes with general purpose tools that have no inherent "economic content," but that can be used in many economic models as building blocks or utilities. Tools might include functions for data analysis (e.g. calculating Lorenz shares from data, or constructing a non-parametric kernel regression), functions to create and manipulate discrete approximations to continuous distributions, or classes for constructing interpolated approximations to non-parametric functions. Tool modules reside in the "top level" of HARK and have names like `HARK.simulation` and `HARK.interpolation`. The core functionality of HARK is in the tools modules; these will be discussed in detail in the following section.

Model modules specify particular economic models, including classes to represent agents in the model and the "market structure" in which they interact, and functions for solving the "one period problem" of those models. For example, `ConsIndShockModel.py` concerns consumption-saving models in which agents have CRRA utility over consumption and face idiosyncratic (**I**ndividual) shocks to permanent and transitory income. The module includes classes for representing "types" of consumers, along with functions for solving (several flavors of) the one period consumption-saving problem. When run, model modules might demonstrate example specifications of their

models, filling in the model parameters with arbitrary values. When `ConsIndShockModel.py` is run, it specifies an infinite horizon consumer with a particular discount factor, permanent income growth rate, coefficient of relative risk aversion and other parameters, who faces lognormal shocks to permanent and transitory income each period with a particular standard deviation; it then solves this consumer's problem and graphically displays the results.⁴ Model modules generally have `Model` in their name. There are two broad types of models solved by HARK, "microeconomic" models and aggregate or "macroeconomic" models. In a microeconomic problem, agents solve their problem taking their environment as a given -- the "macro" environment is fixed exogenously. A macroeconomic problem is typically composed of a number of agents solving their own microeconomic problems, whose interactions affect the macroeconomic environment. Thus the aggregate processes that describe the agents' environment is endogenous to the individual-level decisions made by each agent. The two examples illustrate this in the "microeconomic" and "macroeconomic" sections below.

Application modules use tool and model modules to solve, simulate, and/or estimate economic models *for a particular purpose*. While tool modules have no particular economic content and model modules describe entire classes of economic models, applications are uses of a model for some research purpose. For example, `/SolvingMicroDSOPs/StructEstimation.py` uses a consumption-saving model from `ConsIndShockModel.py`, calibrating it with age-dependent sequences of permanent income growth, survival probabilities, and the standard deviation of income shocks (etc); it then estimates the coefficient of relative risk aversion and shifter for an age-varying sequence of discount factors that best fits simulated wealth profiles to empirical data from the Survey of Consumer Finance. A particular application might have multiple modules associated with it, all of which generally reside in one directory. Particular application modules will not be discussed in this paper further; please see [the GitHub page and associated documentation](#) for references to the application modules.

2. Tool Modules

HARK's root directory contains the following tool modules, each containing a variety of functions and classes that can be used in many economic models, or even for mathematical purposes that have nothing to do with economics. We expect that all of these modules will grow considerably in the near future, as new tools are "low hanging fruit" for contribution to the project.

HARK.core

This module contains core classes used by the rest of the HARK ecosystem. A key goal of the project is to create modularity and interoperability between models, making them easy to combine, adapt, and extend. To this end, the `HARK.core` module specifies a framework for economic models in HARK, creating a common structure for them on two levels that can be called "microeconomic" and "macroeconomic".

Beyond the model frameworks, `HARK.core` also defines a "superclass" called `HARKObject`. When solving a dynamic economic model, it is often required to consider whether two solutions are sufficiently close to each other to warrant stopping the process (i.e. approximate convergence). HARK specifies that classes should have a `distance` method that takes a single

input and returns a non-negative value representing the (generally dimensionless) distance between the object in question and the input to the method. As a convenient default, `HARKObject` provides a “universal distance metric” that should be useful in many contexts.⁵ When defining a new subclass of `HARKObject`, the user simply defines the attribute `distance_criteria` as a list of strings naming the attributes of the class that should be compared when calculating the distance between two instances of that class. See [here](#) for online documentation.

HARK.utilities

The `HARK.utilities` module carries a double meaning in its name, as it contains both utility functions (and their derivatives, inverses, and combinations thereof) in the economic modeling sense as well as utilities in the sense of general tools. Utility functions include constant relative risk aversion (CRRA) and constant absolute risk aversion (CARA). Other functions in `HARK.utilities` include data manipulation tools, functions for constructing discrete state space grids, and basic plotting tools. The module also includes functions for constructing discrete approximations to continuous distributions and manipulating these representations.

HARK.interpolation

The `HARK.interpolation` module defines classes for representing interpolated function approximations. Interpolation methods in `HARK` all inherit from a superclass such as `HARKinterpolator1D` or `HARKinterpolator2D`, wrapper classes that ensure interoperability across interpolation methods. These classes all inherit from `HARKObject`, so that they come equipped with the default distance metric.⁶

HARK.simulation: The `HARK.simulation` module provides tools for generating simulated data or shocks for post-solution use of models. Currently implemented distributions include normal, lognormal, Weibull (including exponential), uniform, Bernoulli, and discrete.

HARK.estimation: Methods for optimizing an objective function for the purposes of estimating a model can be found in `HARK.estimation`. As of this writing, the implementation includes minimization by the Nelder-Mead simplex method, minimization by a derivative-free Powell method variant, and two tools for resampling data (e.g., for a bootstrap). Future functionality will include global search methods, including genetic algorithms, simulated annealing, and differential evolution.

3. Model Modules

Microeconomic models in `HARK` use the `AgentType` class to represent agents with an intertemporal optimization problem. Each of these models specifies a subclass of `AgentType`; an instance of the subclass represents agents who are ex-ante homogeneous (they have common values for all parameters that describe the problem, such as risk aversion). The `AgentType` class has a `solve` method that acts as a “universal microeconomic solver” for any properly formatted model, making it easier to set up a new model and to combine elements from different models; the solver is intended to encompass any model that can be framed as a sequence of one period problems.⁷

Macroeconomic models in `HARK` use the `Market` class to represent a market or other mechanisms by which agents’ (i.e. instances of `AgentType` subclasses) interactions are aggregated

to produce “macro-level” outcomes. For example, the market in a consumption-saving model might combine the individual asset holdings of all agents in the market to generate aggregate savings and capital in the economy, which in turn produces the interest rate that agents care about. Agents then learn the aggregate capital level and interest rate, which affects their future actions. In this way, objects that *microeconomic* agents treat as exogenous when solving their individual-level problems (such as the interest rate) are made *endogenous* at the macroeconomic level through the `Market` aggregator. Like `AgentType`, the `Market` class also has a `solve` method, which seeks out a dynamic general equilibrium rule governing the aggregate processes.

Microeconomics: the AgentType Class

The core of our microeconomic dynamic optimization framework is a flexible object-oriented representation of economic agents. Each microeconomic model defines a subclass of `AgentType`, specifying additional model-specific features and methods while inheriting the methods of the superclass. This section provides a brief example of a problem solved by a microeconomic instance of `AgentType`.

Sample Model: Perfect Foresight Consumption-Saving:

To provide a concrete example of how the `AgentType` class works, consider the very simple case of a perfect foresight consumption-saving model. The agent has time-separable, additive CRRA preferences over consumption C_t , discounting future utility at a constant rate. He receives a particular stream of labor income Y_t each period and knows the interest rate R on assets A_t that he holds from one period to the next. His decision about how much to consume C_t in a particular period out of total market resources M_t can be expressed in Bellman form as:

$$\begin{aligned} V_t(M_t) &= \max_{C_t} u(C_t) + \beta(1 - D_{t+1})E[V_{t+1}(M_{t+1})], \\ A_t &= M_t - C_t, \\ M_{t+1} &= RA_t + Y_{t+1}, \\ Y_{t+1} &= \Gamma_{t+1}Y_t, \\ u(C) &= \frac{C^{1-\rho}}{1-\rho}. \end{aligned}$$

The agent’s problem is thus characterized by values of ρ , R , and β , plus sequences of survival probabilities $(1 - D_{t+1})$ and income growth factors Γ_{t+1} for $t = 0, \dots, T - 1$. This problem has an analytical solution for both the value function and the consumption function.

The `ConsIndShockModel.py` module defines the class `PerfForesightConsumerType` as a subclass of `AgentType` and provides `solver` classes for several variations of a consumption-saving model, including the perfect foresight problem. A `HARK` user could specify and solve a ten period perfect foresight model with the following two commands (the first command is split over multiple lines) :

```
MyConsumer = PerfForesightConsumerType(
    time_flow=True, cycles=1, AgentCount = 1000,
    CRRA = 2.7, Rfree = 1.03, DiscFac = 0.98,
    LivPrb = [0.99,0.98,0.97,0.96,0.95,0.94,0.93,
              0.92,0.91,0.90],
    PermGroFac = [1.01,1.01,1.01,1.01,1.01,1.02,
                  1.02,1.02,1.02,1.02] )
```

```
MyConsumer.solve()
```

The first line makes a new instance of `ConsumerType`, specifies that time is currently “flowing” forward, specifies that the se-

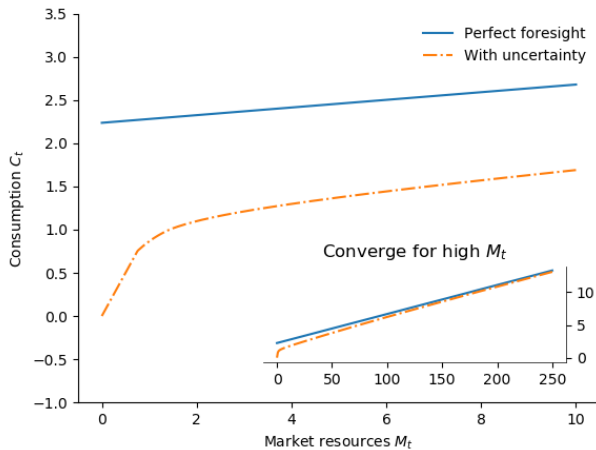


Fig. 1: Consumption Functions

quence of periods happens exactly once, and that, if the model is simulated after it is solved, there are 1000 agents with these exact characteristics. The next five lines (all part of the same command) set the time-invariant (CRRR is ρ , Rfree is R , and DiscFac is β) and time-varying parameters (LivPrb is $(1 - D_{t+1})$, PermGroFac is Γ_{t+1}). After running the solve method, MyConsumer will have an attribute called solution, which will be a list with eleven ConsumerSolution objects, representing the period-by-period solution to the model.⁸

The consumption function for a perfect foresight consumer is a linear function of market resources-- not terribly exciting. The marginal propensity to consume out of wealth doesn't change whether the consumer is rich or poor. When facing *uncertain* income, however, the consumption function is concave: the marginal propensity to consume is very high when agents are poor, and lower when they are rich. Moreover, agents facing income risk save more than agents under certainty. However, as agents facing uncertainty get richer, their consumption function converges to the perfect foresight consumption function-- rich but uncertain agents act like agents who face no income risk. In Figure 1, the solid blue line is consumption under certainty, while the dashed orange line is consumption under uncertainty. The inset plot demonstrates that these two functions converge as the horizontal axis of this plot is extended.

Macroeconomics: the Market Class

The modeling framework of AgentType is called "microeconomic" because it pertains only to the dynamic optimization problem of individual agents, treating all inputs of the problem from their environment as exogenously fixed. In what we label as "macroeconomic" models, some of the inputs for the microeconomic models are endogenously determined by the collective states and choices of other agents in the model. In a rational dynamic general equilibrium, there must be consistency between agents' beliefs about these macroeconomic objects, their individual behavior, and the realizations of the macroeconomic objects or processes that result from individual choices.

The Market class in HARK.core provides a framework for such macroeconomic models, with a solve method that searches for a rational dynamic general equilibrium. An instance of Market includes as an attribute a list of AgentType objects

that compose the economy, a method for transforming microeconomic outcomes (states, controls, and/or shocks) into macroeconomic outcomes, and a method for interpreting a history or sequence of macroeconomic outcomes into a new "dynamic rule" for agents to believe. Agents treat the dynamic rule as an input to their microeconomic problem, conditioning their optimal policy functions on it. A dynamic general equilibrium is a fixed point dynamic rule: when agents act optimally while believing the equilibrium rule, their individual actions generate a macroeconomic history consistent with the equilibrium rule.

Down on the Farm: The Market class uses a farming metaphor to conceptualize the process for generating a history of macroeconomic outcomes in a model. Suppose all AgentType agents in the economy believe in some dynamic rule (i.e. the rule is stored as attributes of each AgentType, which directly or indirectly enters their dynamic optimization problem), and that they have each found the solution to their microeconomic model using their solve method. Further, the macroeconomic and microeconomic states have been reset to some initial orientation.

To generate a history of macroeconomic outcomes, the Market repeatedly loops over the following steps a set number of times:

- 1) sow: Distribute the macroeconomic state variables to all AgentTypes in the market.
- 2) cultivate: Each AgentType executes their marketAction method, often corresponding to simulating one period of the microeconomic model.
- 3) reap: Microeconomic outcomes are gathered from each AgentType in the market.
- 4) mill: Data gathered by reap is processed into new macroeconomic states according to some "aggregate market process".
- 5) store: Relevant macroeconomic states are added to a running history of outcomes.

This procedure is conducted by the makeHistory method of Market as a subroutine of its solve method. After making histories of the relevant macroeconomic variables, the market then executes its calcDynamics function with the macroeconomic history as inputs, generating a new dynamic rule to distribute to the AgentType agents in the market. The process then begins again, with the agents solving their updated microeconomic models given the new dynamic rule; the solve loop continues until the "distance" between successive dynamic rules is sufficiently small.

Each subclass of Market has its own mill and calcDynamics methods, and designates which variables are to be gathered reap and distributed by sow, thus specifying what it means to generate "aggregate outcomes" and "form beliefs" in that particular model. We believe that the Market framework is general enough to encompass a very wide range of disparate models, from standard models in which individual assets are aggregated into productive capital, to models of choice over health insurance contracts with adverse selection and moral hazard, to models of direct agent-to-agent interaction more commonly seen in other scientific fields.

4. Summary and Conclusion

The Econ-ARK project's broadest aim is to provide a platform for improving communication and collaboration among economists on technical and computational questions. Its first framework, the

HARK project, is a modular code library for constructing microeconomic and macroeconomic models with agents who differ from each other in serious ways: in dimensions whose consequences cannot be captured by analyzing the behavior of a single agent with average characteristics.

The HARK project is the starting point because it is an area where both the need and opportunities for improvement are great. In particular, existing code to solve HA models tends to be bespoke and idiosyncratic, with the consequence that tools are often reinvented by different researchers working on similar problems. Researchers should spend their valuable time producing research, not reinventing wheels. The HARK toolkit already provides a useful set of industrial strength, reliable, reusable wheels, constructed using a simple and easily extensible framework with clear documentation and testing regimens.

Part of the reason we are confident our goal is feasible is that the tools now available – Python, GitHub, and Jupyter notebooks among them – have finally reached a stage of maturity that can handle the communication of almost any message an economist might want to convey.⁹

The longer-term goals of the Econ-ARK project are to create a collaborative codebase that can serve the entire discipline of economics, employing the best of modern software development tools to accelerate understanding and implementation of cutting edge research tools. The solution methods employed in HARK are not the only methods available, and those who have additional methodological suggestions are strongly encouraged to contribute. The interested user should check the Econ-ARK GitHub page, particularly the [HARK sub-page](#). There you will find a README and documentation. For the interested contributor, the [issues page](#) outlines the future improvements in progress. Issues labeled with "help wanted" are particularly good for getting started with contributing.

Acknowledgements

The Econ-ARK project is supported by a generous grant from the Alfred P. Sloan Foundation, with fiscal sponsorship from NumFOCUS. The authors would like to thank both organizations for their time, resources, and expertise.

Bibliography

- [Adjemian2011] Adjemian, Stephane, Houtan Bastani, Michel Juillard, Ferhat Mihoubi, George Perendia, Marco Ratto, and Sebastien Villemot. 2011. "Dynare: Reference Manual, Version 4." *Dynare Working Papers* 1, CEPREMAP. [RePEc: cpmdynare/001](#).
- [Carroll2012] Carroll, Christopher. 2012. "Solving Microeconomic Dynamic Stochastic Optimization Problems." *Lecture Notes, Johns Hopkins University*. [url](#)
- [Carroll2017a] Carroll, Christopher, Jiri Slacalek, Kiichi Tokuoka, and Matthew N White. 2017. "The Distribution of Wealth and the Marginal Propensity to Consume." *Quantitative Economics* 8 (3). Wiley Online Library: 977–1020. [doi:10.3982/QE694](#)
- [Carroll2017b] Carroll, Christopher, Alexander Kaufman, David Low, Nathan Palmer, and Matthew White. 2017. "A User's Guide for Hark: Heterogeneous Agents Resources and toolkit." *Econ ARK*. [url](#)
- [Geanakoplos2010] Geanakoplos, John. 2010. "The Leverage Cycle." *NBER Macroeconomics Annual* 24 (1). The University of Chicago Press: 1-66. [doi:10.1086/648285](#)

- [Geanakoplos2012] Geanakoplos, John, Robert Axtell, J Doyne Farmer, Peter Howitt, Benjamin Conlee, Jonathan Goldstein, Matthew Hendrey, Nathan M. Palmer, and Chun-Yi Yang. 2012. "Getting at Systemic Risk via an Agent-Based Model of the Housing Market." *American Economic Review* 102 (3): 53-58. [doi:10.1257/aer.102.3.53](#)

- [Guvenen2011] Guvenen, Fatih. 2011. "Macroeconomics with Heterogeneity: A Practical Guide," *Economic Quarterly, Federal Reserve Bank of Richmond* 97 (3): 255-326. [doi:10.3386/w17622](#)

- [Hommes2018] Hommes, Cars, and Blake LeBaron, eds. 2018. "Handbook of Computational Economics, Vol 4: Heterogeneous Agent Modeling," *Handbook of Computational Economics*, Elsevier, Vol 4: 2-796. [doi:10.1016/S1574-0021\(18\)30018-2](#)

- [Kaplan2018] Kaplan, Greg, Benjamin Moll, and Giovanni L. Violante. 2018. "Monetary Policy According to HANK." *American Economic Review* 108 (3): 697-743. [doi:10.1257/aer.20160042](#)

- [Kirman1992] Kirman, Alan P. 1992. "Whom or What Does the Representative Individual Represent?" *Journal of Economic Perspectives* 6 (2): 117-136. [doi:10.1257/jep.6.2.117](#)

- [Tsfatsion2006] Tsfatsion, Leigh, Kenneth L. Judd, eds. 2006. "Handbook of Computational Economics, Vol 2: Agent-Based Computational Economics," *Handbook of Computational Economics*, Elsevier, Vol 2: 829-1660. [doi:10.1016/S1574-0021\(05\)02039-3](#)

1. In this context, "heterogeneity" refers to both ex post heterogeneity--agents attaining different states or making different choices because they have experienced different random shocks in the model-- and ex ante heterogeneity--agents differing in their preferences, beliefs, or other innate attribute before the model "begins".

2. It is possible that some of the foundational tools from QuantEcon could be incorporated into the Econ-ARK, with the permission of its project leads. Our teams are in communication, and their advice has been valuable.

3. HARK does not impose the assumption of rationality; we use it here for exposition because it is the standard assumption in economics. The modular structure of the toolkit makes it easy to remove this assumption by, e.g., having agents misperceive their own problem, imperfectly process information, or form beliefs about aggregate processes that are not "justified" by the history.

4. Running `ConsIndShockModel.py` also demonstrates other variations of the consumption-saving problem, but their description is omitted here for brevity.

5. Roughly speaking, the universal distance metric is a recursive supnorm, returning the largest distance between two instances, among attributes named in `distance_criteria`. Those attributes might be complex objects themselves rather than real numbers, generating a recursive call to the universal distance metric.

6. Interpolation methods currently implemented in HARK include (multi)linear interpolation up to 4D, 1D cubic spline interpolation, 2D curvilinear interpolation over irregular grids, a 1D "lower envelope" interpolator, and others.

7. See [Carroll2017b] for a much more thorough discussion.

8. The solution to a dynamic optimal control problem is a set of policy functions and a value function, for each period. The policy function for this consumption-saving problem is how much to consume C_t for a given amount of market resources M_t . The eleventh and final element of `solution` represents the trivial solution to the terminal period of the problem. For a much more detailed discussion, please see [Carroll2017b].

9. See the recent blog post by Paul Romer, "[Jupyter, Mathematica, and the Future of the Research Paper](#)" for a fuller argument).

Developing a Start-to-Finish Pipeline for Accelerometer-Based Activity Recognition Using Long Short-Term Memory Recurrent Neural Networks

Christian McDaniel^{‡*}, Shannon Quinn[‡]



Abstract—Increased prevalence of smartphones and wearable devices has facilitated the collection of triaxial accelerometer data for numerous Human Activity Recognition (HAR) tasks. Concurrently, advances in the theory and implementation of long short-term memory (LSTM) recurrent neural networks (RNNs) has made it possible to process this data in its raw form, enabling on-device online analysis. In this two-part experiment, we have first amassed the results from thirty studies and reported their methods and key findings in a meta-analysis style review. We then used these findings to guide our development of a start-to-finish data analysis pipeline, which we implemented on a commonly used open-source dataset in a proof of concept fashion. The pipeline addresses the large disparities in model hyperparameter settings and ensures the avoidance of potential sources of data leakage that were identified in the literature. Our pipeline uses a heuristic-based algorithm to tune a baseline LSTM model over an expansive hyperparameter search space and trains the model on standardized windowed accelerometer signals alone. We find that we outperform other baseline models trained on this data and are able to compete with benchmark results from complex models trained on higher-dimensional data.

Index Terms—Neural Network, Human Activity Recognition, Recurrent Neural Network, Long Short-Term Memory, Accelerometer, Machine Learning, Data Analysis, Data Science, Hyperparameter Optimization, Hyperparameter

Introduction

Human Activity Recognition (HAR) is a time series classification problem in which a classifier attempts to discern distinguishable features from movement-capturing on-body sensors [KHC10]. The most common sensor for HAR tasks is the accelerometer, which measures high-frequency (30-200Hz) triaxial time series recordings, often containing noise, imprecision, missing data, and long periods of inactivity between meaningful segments [RDML05], [BI04], [OR16]. Consequently, attempts to use traditional classifiers typically require significant preprocessing and technical engineering of hand crafted features from raw data, resulting in a barrier to entry for the field and making online and on-device data processing impractical [GRX16], [MS10], [GBGG16], [RDML05], [OR16].

* Corresponding author: clm121@uga.edu

‡ University of Georgia

The limitations of classical methods in this domain have been alleviated by concurrent theoretical and practical advancements in artificial neural networks (ANNs), which are more suited for complex non-linear data. While convolutional neural networks (CNNs) are attractive for their automated feature extraction capabilities during convolution and pooling operations [SS17], [REBS17], [FFH⁺16], [SKP18], [ZSO17], [GRX16], [OR16], [GBGG16], recurrent neural networks (RNNs) are specifically designed to extract information from time series data due to the recurrent nature of their data processing and weight updating operations [WZ89]. Furthermore, whereas earlier implementations of RNNs experienced problems when processing longer time series (tens to hundreds of time steps), the incorporation of a multi-gated memory cell in long short-term memory recurrent neural networks (LSTMs) [HS97] along with other regularization schemes helped alleviate these issues.

As RNN usage continues, numerous studies have emerged to address various aspects of understanding and implementing these complex models, namely regarding the vast architectural and hyperparameter combinations that are possible [GSS02], [RG17], [PW17], [KJFF15], [MKS17]. Unfortunately, these pioneering studies tend to focus on tasks other than HAR, leaving the time series classification tasks of HAR without domain-specific architecture guidance.

In a meta-analysis style overview of the use of LSTM RNNs for HAR experiments across 30 reports (discussed below), we found a general lack of consensus regarding the various model architectures and hyperparameters used. Often, a given pair of experiments explored largely or entirely non-overlapping ranges for a single hyperparameter. Key architectural and procedural details are often not included in the reports, making reproducibility impossible. The analysis pipelines employed are often lacking detail and sources of data leakage, where information from the testing data is exposed to the model during training, appear to be overlooked in certain cases. Without clear justifications for model implementations and deliberate, reproducible data analysis pipelines, objective model comparisons and inferences from results cannot be made. For these reasons, the current report seeks to summarize the previous implementations of LSTMs for HAR research available in literature and outline a structured data analysis pipeline for this domain. We implement a truncated version of our pipeline, optimizing a baseline LSTM over an expansive hyperparameter search space, and obtain results on par

with benchmark studies. We suspect that our efforts will encourage scientific rigor in the field going forward and initiate more granular exploration of the field as we understand these powerful data analysis tools within this domain.

Background

This section is intended to give the reader a digestible introduction to ANNs, RNNs, and the LSTM cell. The networks will be discussed as they relate to multi-class classification problems as is the task in HAR.

Artificial Neural Networks The first ANN architecture was proposed by Drs. Warren McCulloch and Walter Pitts in 1943 as a means to emulate the cumulative semantic functioning of groups of neurons via propositional logic [MP43], [Ger17]. Frank Rosenblatt subsequently developed the Perceptron in 1957 [Ros57]. This ANN variation carries out its step-wise operations via mathematical constructs known as linear threshold units (LTUs). The LTU operates by aggregating multiple weighted inputs and feeding this summation u through an activation function $f(u)$ or step function $\text{step}(u)$, generating an interpretable output \tilde{y} (e.g. 0 or 1) [Ger17].

$$\begin{aligned}\tilde{y} &= f(u) \\ &= f(w \cdot x)\end{aligned}$$

where \cdot is the dot product operation from vector calculus. x is a single instance of the training data, containing values for all n attributes of the data. As such, w is also of length n , and the entire training data set for all m instances is a matrix X of dimensions m by n (i.e., $m \times n$).

A 2-layer ANN can be found in Figure 1 A. Each attribute in instance $x(i)$ represents a node in the perceptron's input layer, which simply provides the raw data to the the output layer - where the LTU resides. To represent k target classes, k LTU nodes are included in the output layer, each corresponding to a single class in y . Each LTU's prediction \tilde{y} indicates the predicted probability that the training instance belongs to the corresponding class. The LTU output with the largest value - $\max(\tilde{y})$ - is taken as the overall predicted class for the instance of the data being analyzed. Taken over the entire dataset, each LTU has a prediction vector \tilde{y}_k length m and the entire output layer produces a prediction matrix \tilde{Y} with dimensions $m \times k$. Additionally, each LTU contains its own weight vector w_k of length n (i.e., a fully-connected network), resulting in a weight matrix W of dimensions $n \times k$.

ANNs often contain complex architectures with additional layers, which allow for nonlinear transformations of the data and increase the flexibility and robustness of the model. If we look at a simple three-layer neural network (see Figure 1 B), we see input and output layers as described above, as well as a layer in the middle, termed a *hidden layer*. This layer acts much like the output layer, except that its outputs z for each training instance are fed into the output layer, which then generates predictions \tilde{y} from z alone. The complete processing of all instances of the dataset, or all instances of a portion of the dataset called a *mini-batch*, through the input layer, the hidden layer, and the output layer marks the completion of a single *forward pass*.

For the model to improve, the outputs generated by this forward pass must be evaluated and the model updated in an attempt to improve the model's predictive power on the data. An error term (e.g., sum of squared error (*sse*)) is calculated by comparing individual predictions \tilde{y}_k to corresponding ground truth target values in y_k . Thus, an error matrix E is generated containing

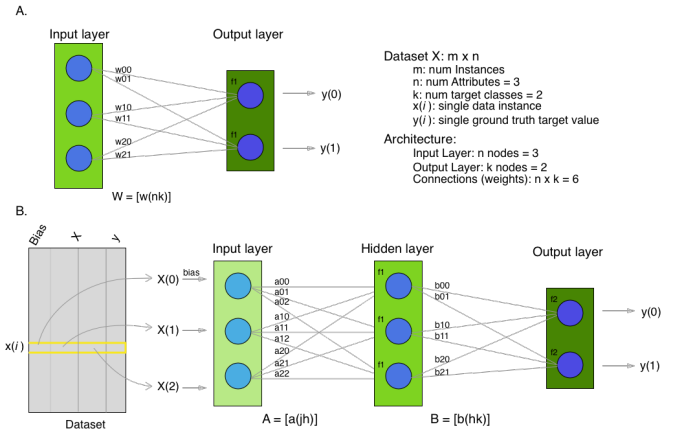


Fig. 1: A. A two-layer network and associated dimensions of the components. B. A three-layer network showing a single data instance $x(i)$ being fed in as input.

error terms over all k classes for all m training instances. This error matrix is used as an indicator for how to adjust the weight matrix in the output layer so as to yield more accurate predictions, and the corrections made to the output layer give an indication of how to adjust the weights in the hidden layer. This process of carrying the error backward from the output layer through the hidden layer(s) is known as *backpropagation*. One forward pass and subsequent backpropagation makes up a single *epoch*, and the training process consists of many epochs repeated in succession to iteratively improve the model.

The iterative improvements to the model are known as *optimization*, and many methods exist to carry this process out. The common example is stochastic gradient descent (SGD), which calculates the gradient of the error - effectively the steepness of E 's location as it "descends" toward lower error - and adjusts the weight matrices at each layer in a direction opposite this gradient. The change to be applied to the weight matrices is mediated via a learning rate η [Mil18].

$$E = Y - f(XW)$$

optimization:

$$\min_W \|E\|_F$$

$$hsse_W = \frac{1}{2} \sum_{c=0}^{k-1} (y_c - f(X \cdot w_c)) \cdot (y_c - f(X \cdot w_c))$$

$$\frac{\partial hsse}{\partial w_k} = X * [f'(X \cdot w_k) * e_k] * \eta = -X * \delta_k * \eta$$

where $f(\dots)$ represents the activation function, \min_W represents the objective function of minimizing with respect to W , and $\|E\|_F$ stands for the Frobenius norm on the error matrix E . $hsse_W$ represents the halved (for mathematical convenience) sum of squared error, calculated for all k nodes in the output layer. $f'(\dots)$ represents the derivative of the activation function over the term in the parentheses.

Looking at our three-layer neural network depicted in Figure 1, a single epoch would proceed as follows:

- 1) Conduct a forward pass, compute \tilde{y} and compare with y to generate the error term:

$$z_h = f_1(a_h \cdot x)$$

$$\tilde{y}_k = f_2(b_k \cdot z)$$

$$e_k = y_k - \tilde{y}_k$$

- 2) Backpropagate the error regarding the correction needed for \tilde{y} .
- 3) Backpropagate the correction to the hidden layer.
- 4) update weight matrices A and B via δ^y and δ^z :

$$\begin{aligned} b_{hk} &= b_{hk} - z_h \delta_k^y * \eta \\ &= b_{hk} - \frac{\partial h_{sse}}{\partial b_{hk}} * \eta \end{aligned}$$

$$\begin{aligned} a_{jh} &= a_{jh} - x_j \delta_h^z * \eta \\ &= a_{jh} - \frac{\partial h_{sse}}{\partial a_{jh}} * \eta \end{aligned}$$

sse is commonly used as the error term for regression problems, whereas squared error or *cross entropy* is typical for classification problems.

$$\text{cross entropy} = - \sum_{i=1}^m \sum_{c=1}^k y_{ic} * \log(f_c(x_i))$$

The high flexibility of neural networks increases the chances of overfitting, and there are various ways to avoid this. *Early stopping* is a technique that monitors the change in performance on a validation set (subset of the training set) and stops training once improvement slows sufficiently. *Weight decay* helps counter large updates to the weights during backpropagation and slowly shrinks the weights toward zero in proportion to their relative sizes. Similarly, the *dropout* technique "forgets" a specified proportion of the outputs from a layer's neurons by not passing those values on to the next layer. *Standardizing* the input is important, as it encourages all inputs to be treated equally during the forward pass by scaling and mitigating outliers' effects [Ger17], [Mil18].

Other hyperparameters tend to affect training efficiency and effectiveness and tend to differ with different datasets and types of data. Hammerla, et. al. found *learning rate* η to be an important hyperparameter in terms of its effect on performance [HHP16]. Too small a learning rate and the model will exhibit slow convergence during training, while too large a value will lead to wild oscillations during optimization [Mil18]. Hammerla, et. al. also find the *number of units* per layer n to be important, and Miller adds that too many hidden units is better than too few, leading to sparse layers of weight matrices versus restricting flexibility of the model, respectively. *Bias* helps account for irreducible error in the data and is implemented via a node whose inputs are always 1's (top node in the input layer of Figure 1 A). Reimers and Gurevych emphasize the importance of weight initialization for model performance in their survey of the importance of hyperparameter tuning for using LSTMs for language modeling [RG17]. Jozefowicz, et. al. cite the initialization of the forget gate bias to 1 as a major factor in LSTM performance [JZS15].

Recurrent Neural Networks (RNNs) The recurrent neuron, developed by Drs. Ronald Williams and David Zipser in 1989 [WZ89], is extremely useful in training a model on sequence data. Recurrent neurons address temporal dependencies along the temporal dimension of time series data by sending their outputs both forward to the next layer and "backward through time," looping the neuron's output back to itself as input paired with new input from the previous time step. Thus, a component of the input to the neuron is an accumulation of activated inputs from each

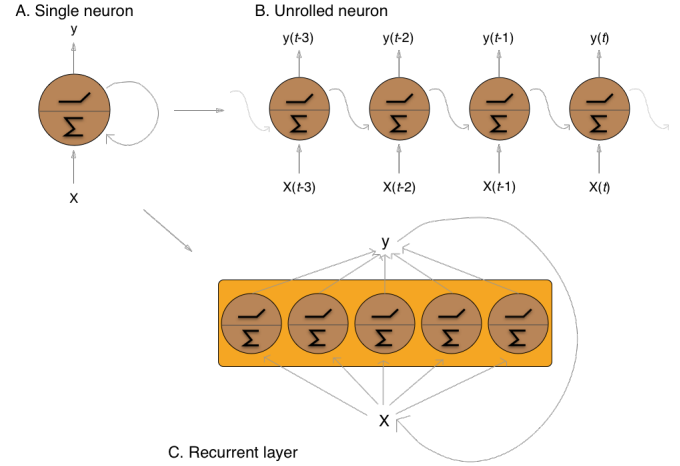


Fig. 2: The recurrent neuron from three perspectives. **A.** A single recurrent neuron, taking input from X , aggregating this input over all timesteps in a summative fashion and passing the summation through an activation function at each timestep. **B.** The same neuron unrolled through time, making it resemble a multilayer network with a single neuron at each layer. **C.** A recurrent layer containing five recurrent nodes, each of which processes the entire dataset X through all time point.

previous time step. Figure 2 depicts a recurrent neuron as part of a recurrent layer. Recurrent layers are placed between input layers and output layers and can be used in succession with densely connected and convolutional layers.

Instead of a single weight vector as in ANN neurons, RNN neurons have two sets of weights, one (w_x) for the new inputs x_t and one (w_y) for the outputs of the previous time step $y_{(t-1)}$, yielding matrices W_x and W_y when taken over the entire layer. The portion of the neuron which retains a running record of the previous time steps is the *memory cell* or just the *cell* [Ger17].

Outputs of the recurrent layer:

$$y_{(t)} = \phi(W_x \cdot x_{(t)} + W_y \cdot Y_{(t-1)} + b)$$

where ϕ is the activation function and b is the bias vector of length n (the number of neurons).

The *hidden state*, or the *state*, of the cell ($h_{(t)}$) is the information that is kept in memory over time.

To train these neurons, we "unroll" them after a complete forward pass to reveal a chain of linked cells the length of time steps t in a single input. We then apply standard backpropagation to these links, calling the process backpropagation through time (BPTT). This works relatively well for very short time series, but once the number of time steps increases to tens or hundreds of time steps, the network essentially becomes very deep during BPTT and problems arise such as very slow training and exploding and vanishing gradients [Ger17]. Various hyperparameter and regularization schemes exist to alleviate exploding/vanishing gradients, including *gradient clipping* [PMB13], *batch normalization*, *dropout*, and the long short-term memory (LSTM) cell originally developed by Sepp Hochreiter and Jurgen Schmidhuber in 1997 [HS97].

Long Short-Term Memory (LSTM) RNNs The LSTM cell achieves faster training and better long-term memory than vanilla RNN neurons by maintaining two state vectors, the short-term

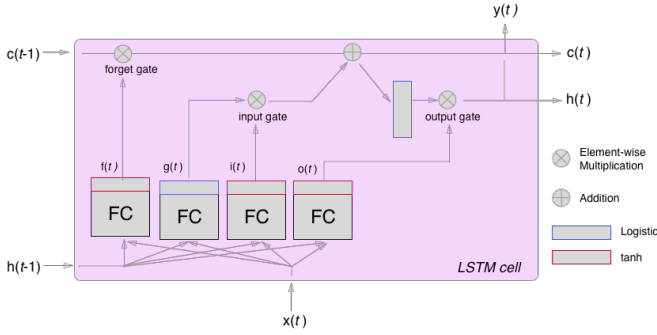


Fig. 3: The inner mechanisms of an LSTM cell. From outside the cell, information flows similarly as with a vanilla recurrent cell, except that the state now exists as two parts, one for long-term memory ($c_{(t)}$) and the other for short-term memory ($h_{(t)}$). Inside the cell, four different sub-layers and associated gates are revealed.

state $h_{(t)}$ and the long-term state $c_{(t)}$, mediated by a series of inner gates, layers, and other functions. These added features allow the cell to process the time series in a deliberate manner, recognizing meaningful input to store long-term and later extract when needed, and forget unimportant information or that which is no longer needed [Ger17].

As can be seen in Figure 3, when the forward pass advances by one time step, the new time step’s input enters the LSTM cell and is copied and fed into four independent fully-connected layers (each with its own weight matrix and bias vector), along with the short-term state from the previous time step, $h_{(t-1)}$. The main layer is $g_{(t)}$, which processes the inputs via \tanh activation function. In the basic recurrent cell, this is sent straight to the output; in the LSTM cell, part of this is incorporated in the long-term memory as decided by the *input gate*. The input gate also takes input from another layer, $i_{(t)}$, which processes the inputs via the sigmoid activation function σ (as do the next two layers). The third layer, $f_{(t)}$, processes the inputs, combines them with $c_{(t-1)}$, and passes this combination through a *forget gate* which drops a portion of the information therein. Finally, the fourth fully-connected layer $o_{(t)}$ processes the inputs and passes them through the *output gate* along with a copy of the updated long-term state $c_{(t)}$ after its additions from $f_{(t)}$, deletions by the forget gate, further additions from the filtered $g_{(t)}-i_{(t)}$ combination and a final pass through a \tanh activation function. The information that remains after passing through the output gate continues on as the short-term state $h_{(t)}$.

$$i_{(t)} = \sigma(W)xi \cdot x_{(t)} + W_{hi} \cdot h_{(t-1)} + b_i$$

$$f_{(t)} = \sigma(W)xf \cdot x_{(t)} + W_{hf} \cdot h_{(t-1)} + b_f$$

$$o_{(t)} = \sigma(W)xo \cdot x_{(t)} + W_{ho} \cdot h_{(t-1)} + b_o$$

$$g_{(t)} = \sigma(W)xg \cdot x_{(t)} + W_{hg} \cdot h_{(t-1)} + b_g$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})$$

where \otimes represents element-wise multiplication [Ger17].

Related Works

The following section outlines the nuanced hyperparameter combinations used by 30 studies available in literature in a meta-analysis style survey. Published works as well as pre-published and academic research projects were included so as to gain insight into the state-of-the-art methodologies at all levels and increase the volume of works available for review. It should be noted that the following summaries are not necessarily entirely exhaustive regarding the specifications listed. Additionally, many reports did not include explicit details of many aspects of their research.

The survey of previous experiments in this field provided blueprints for constructing an adequate search space of hyperparameters. We have held our commentary on the findings of this meta-study until the Discussion section.

Experimental Setups Across the 30 studies, each used a unique implementation of LSTMs for the research conducted therein. Data sets used include the OPPORTUNITY Activity Recognition dataset [OR16], [RVCK17], [GRX16], [ZYCG17], [Bro17], [GP17], UCI HAR dataset [U18], [ZYCG17], PAMAP2 [OR16], [Set18], [GP17], [ZYH⁺18], Skoda [OR16], [GP17], WISDM [CZZZ16], [U18], and various study-specific and/or internally-collected datasets [MMB⁺18]. Activity classes include “Activities of Daily Life” (ADL; e.g., opening a drawer, climbing stairs, walking, or sitting down), smoking [Ber17], cross-country skiing [REBS17], eating [KDD17], nighttime scratching [MAR⁺16], driving [CFF⁺17], and so on.

Data analysis pipelines employed include cross validation [LBMG15], repeating trials [SS16], and various train-validation-test splitting procedures [SS17], [WA17], [HDJS18]. Most studies used the Python programming language and implemented LSTMs via third-party libraries such as Theano Lasagne, RNNLib, and Keras with TensorFlow.

Preprocessing Some reports kept preprocessing to a minimum, e.g., linear interpolation to fill missing values [OR16], per-channel normalization [OR16], [HDJS18], and standardization [CZZZ16], [ZYCG17]. Zhao, et. al. standardized the data to have 0.5 standard deviation [ZYCG17] as opposed to the typical unit standard deviation, citing Wiesler, et. al. as supporting this nuance for deep learning implementations [WRSN14].

More advanced noise reduction strategies include kernel smoothing [GRX16], removing the gravity component [MAR⁺16], applying a low-pass filter [LBMG15], removing the initial and last 0.5 seconds [HDJS18]. Moreau, et. al. grouped together segments of data from different axes, tracking the dominant direction of motion across axes [MAR⁺16].

For feeding the data into the models, the sliding window technique was commonly used, with window sizes ranging from 32 [MMB⁺18] to 5000 [ZYCG17] milliseconds (ms); typically 50% of the window size was used as the step size [REBS17], [SS17], [Bro17], [OR16]. Guan and Plotz ran an ensemble of models, each using a random sampling of a random number of frames with varying sample lengths and starting points. This method is similar to the bagging scheme of random forests and was implemented to increase robustness of the model [GP17].

Architectures Numerous architectural and hyperparameter choices were made among the various studies. Most studies used two LSTM layers [OR16], [CZZZ16], [KDD17], [RVCK17], [U18], [ZYCG17], [GP17], [HDJS18], [MMB⁺18], while others used a single layer [WA17], [Bro17], [SS16], [CFF⁺17], [ZWYM16], [ZYH⁺18], [SKP18], three layers [ZWYM16], or four layers [MP17].

The number of units (i.e., nodes, LSTM cells) per layer range from 3 [MAR⁺16] to 512 [Set18]. Several studies used different numbers of units for different circumstances – e.g., three units per layer for unilateral movement (one arm) and four units per layer for bilateral movement (both arms) [MAR⁺16] or 28 units per layer for the UCI HAR dataset (lower dimensionality) versus 128 units per layer for the Opportunity dataset [ZYCG17]. Others used different numbers of units for different layers of the same model – e.g., 14-14-21 for a 3-layer model [ZWYM16].

Almost all of the reports used the sigmoid activation for the recurrent connections within cells and the tanh activation function for the LSTM cell outputs, as these are the activation functions used the original paper [HS97]. Other activation functions used for the cell outputs include ReLU [ZYCG17], [HDJS18] and sigmoid [ZYH⁺18].

Several studies designed or utilized novel LSTM architectures that went beyond the simple tuning of hyperparameters. Architectures tested include the combination of CNNs with LSTMs such as ConvLSTM [GRX16], DeepConvLSTM [OR16], [SS17], [Bro17], and the multivariate fully convolutional LSTM network (MLSTM-FCN) [KMDH18]; innovations regarding the connections between hidden units including the bidirectional LSTM (b-LSTM) [REBS17], [Bro17], [MAR⁺16], [LBMG15], [HHP16], hierarchical b-LSTM [LC12], deep residual b-LSTM (deep-res-bidir LSTM) [ZYCG17], and LSTM with peephole connections (p-LSTM) [REBS17]; and other nuanced architectures such as ensemble deep LSTM [GP17], weighted-average spatial LSTM (WAS-LSTM) [ZYH⁺18], deep-Q LSTM [SKP18], the multivariate squeeze-and-excite fully convolutional network ALSTM (MALSTM-FCN) [KMDH18], and similarity-based LSTM [FFH⁺16]. Note that the term “deep” indicates the use of multiple layers of hidden connections - generally three or more LSTM layers qualifies as “deep”.

The use of densely-connected layers before or after the LSTM layers was also common. Kyritsis, et. al. added a dense layer with ReLU activation after the LSTM layers, Zhao, et. al. included a dense layer with tanh activation after the LSTMs, and Musci, et. al. used a dense layer before and after its two LSTM layers [KDD17], [ZWYM16], [MMB⁺18]. The WAS-LSTM, deep-Q LSTM, and the similarity-based LSTM used a combination of dense and LSTM hidden layers.

Training Weight initialization strategies employed include random orthogonal initialization [OR16], [SS17], fixed random seed [Set18], the Glorot uniform initialization [Bro17], random uniform initialization on [-1, 1] [MAR⁺16], or using a random normal distribution [HDJS18]. For mini-batch training, reported batch sizes range from 32 [RVCK17], [Set18] to 450 [Ber17] training examples (e.g., windows) per batch.

Loss functions for monitoring training include categorical cross-entropy [OR16], [MP17], [CZZZ16], [SS17], [KDD17], [Set18], [Bro17], [HDJS18], [ZYH⁺18], F1 score loss [GP17], mean squared error (MSE) [CFF⁺17], and mean absolute error [ZWYM16]. During back propagation, various updating rules – e.g. RMSProp [OR16], [Set18], [Bro17], Adam [MP17], [KDD17], [Bro17], [HDJS18], [ZYH⁺18], and Adagrad [SS16], [HHP16] – and learning rates – 10^{-7} [SS16], 10^{-4} [SS17], [GP17], $2e^{-4}$ [MAR⁺16], $5e^{-4}$ [LBMG15], and 10^{-2} [OR16] are used.

Regularization techniques employed include weight decay of 90% [OR16], [SS17]; update momentum of 0.9 [MAR⁺16], 0.2 [LBMG15], or the Nesterov implementation [SS16]; dropout (e.g.,

50% [OR16], [SS17] or 70% [ZWYM16]) between various layers; batch normalization [ZYCG17]; or gradient clipping using the norm [ZYCG17], [HDJS18], [ZYH⁺18]. Broome chose to test the stateful configuration for its baseline LSTM [Bro17]. In this configuration, unit memory cell weights are maintained between each training example instead of resetting them to zero after each forward pass.

The number of epochs specified ranged from 100 [Bro17] to 10,000 [HDJS18]. Many studies chose to use early stopping to prevent overfitting [JWHT17]. Various patience schemes, specifying how many epochs with no improvement above a given threshold the model should allow, were chosen.

Performance Measures Various performance measures were used to assess the performance of the model, including the F1 score - used by most [OR16], [Bro17], [GRX16], [ZYCG17], [Bro17], classification error [REBS17], accuracy [SS17], [Set18], and ROC [MAR⁺16], [HDJS18].

As this meta-analysis style overview has shown, there are many different model constructions being employed for HAR tasks. The work by the aforementioned studies as well as others have laid the groundwork for this field of research.

Experimental Setup

We implemented a truncated version of our Pipeline, and have made code available for running the entire Pipeline on the UCI HAR Dataset at <https://github.com/xtianmcd/accelstm>.

Data Although many studies use the gyroscope- and magnetometer-supplemented records from complex inertial signals, accelerometer data is the most ubiquitous modality in this field and training models on this data alone helps illuminate the robustness of the model and requires lower computational complexity (i.e., more applicable to online and on-device classifications). As such, this report trains its models on triaxial accelerometer data alone.

The primary dataset used for our experiments is the Human Activity Recognition Using Smartphones Data Set (UCI HAR Dataset) from Anguita, et. al. [AGO⁺13].

UCI HAR Dataset Classes (6) include walking, climbing stairs, descending stairs, sitting, standing, and laying down. Data was collected from built-in accelerometers and gyroscopes (not used in our study) in smartphones worn on the waists of participants.

A degree of preprocessing was applied to the raw signals themselves by the data collectors. The accelerometer data (recorded at 50Hz) was preprocessed to remove noise by applying a third order low pass Butterworth filter with corner frequency of 20Hz and a median filter. A second filter was then applied to the total accelerometer signal (T) to remove the gravity component, leaving the isolated body accelerometer signal (B). The accelerometer signals for both B and T were provided as pre-split single-axis windowed signals divided into separate files; see Figure 4 A. Windows contained 2.56 seconds (128 time steps) of data and had a step size of 50% of the window size. A 70:30 train-to-test split was used, splitting one of the participants between the two sets.

Preprocessing We kept preprocessing to a minimum. We first attempted to “undo” as much of the preprocessing already performed on the data and reformat the data for feeding it into the network. We did this to establish a baseline format for the data at the start of the Pipeline so that data from different datasets can be used. The code for this procedure can be

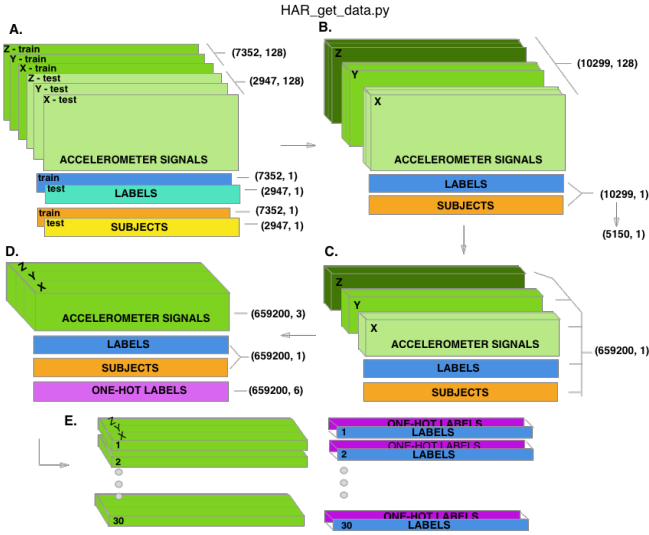


Fig. 4: Depiction of the "undoing" procedure to return the data in the UCI HAR Dataset to its unprocessed form. **A.** Data is provided as train/test-split single-axis windowed accelerometer signals. **B.** Combine train and test sets. **C.** Remove windows; reformat labels and subject include's accordingly. **D.** Axes are combined into a three-dimensional time series; one-hot labels are generated. **E.** 3-D time series and labels are grouped by subject to emulate subject-wise data acquisition.

found in the GitHub repository linked above in the file `accelstm/src/data/HAR_get_data.py`. First, we re-combined the training and testing sets (Figure 4 B). We effectively removed the windows by concatenating together time points from every other window, reforming contiguous time series (Figure 4 C). We then combined each axis-specific time series to form the desired triaxial data format, where each time point consists of the accelerometer values along the x-, y-, and z-axes as a 3-dimensional array (Figure 4 D). We generated one-hot labels in that step as well. We kept track of the participant to which each record belonged (Figure 4 E) so that no single participant was later included in both training and testing sets.

We used an 80:20 training-to-testing split (Figure 5 A-D), and *subsequently* standardized the data by first fitting the standardization parameters (i.e., mean and standard deviation) to the training data and then using these parameters to standardize the training and testing sets separately (Figure 5 E1). This sequenced procedure prevents exposing any summary information about the testing set to the model before training, i.e., data leakage. Finally, a fixed-length sliding window was applied (Figure 5 E2), the windows were shuffled to avoid localization during training (Figure 5 F), and the data was ready to feed into the LSTM neural network.

Training All model training code can be found in the GitHub repository linked above in the folder `accelstm/src/models`. Training the model was broken up into two sections, the first of which consisted of hyperparameter optimization. We employed a heuristic-based search, namely the tree-structured Parzen (TPE) expected improvement (EI) algorithm, in order to more efficiently navigate the vast hyperparameter search space. EI algorithms estimate the ability of a supposed model x to outperform some performance standard y^* , and TPE aims to assist this expectation by heuristically modeling the search space without requiring

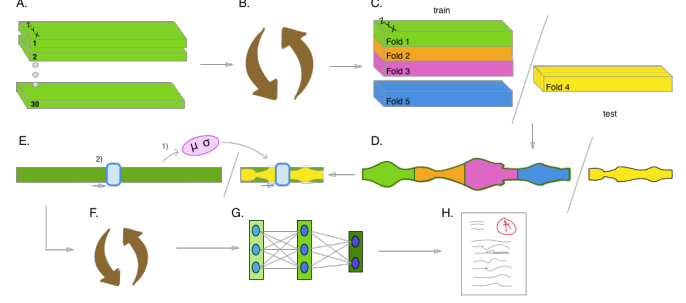


Fig. 5: Outline of the proposed data analysis pipeline. **A.** The data should start as raw tri-axial data files separated into individual records; one record per individual. **B.** Shuffle the records. **C.** Partition the records into k equal groupings for the k -fold cross validation. **D.** Concatenate the records end-to-end within the train and test sets (for feeding in to the LSTM). **E.** Standardize the data, careful to avoid data leakage; subsequently window the data. **F.** Shuffle the windowed data sets. **G.** If in Part 1 of the Pipeline, optimize the model's hyperparameters; if in Part 2, train the optimized model on the training data. **H.** Predict outcomes for the testing data using the trained model and score the results.

exhaustive exploration thereof. TPE iteratively substitutes equally-weighted prior distributions over hyperparameters with Gaussians centered on the examples seen over time. This re-weighting of the search space allows TPE to estimate $p(y)$ and $p(x|y)$ - regarding the performance y from suggested model x - ultimately allowing the EI algorithm to estimate $p(y|x)$ of model M via Bayes Theorem [BBBK11].

$$EI_{y^*}(x) := \int_{-\infty}^{\infty} \max(y^* - y, 0) p_M(y|x) dy$$

becomes

$$\begin{aligned} EI_{y^*}(x) &= \int_{-\infty}^{y^*} \max(y^* - y, 0) p_M(y|x) dy \\ &= \int_{-\infty}^{y^*} \frac{p(x|y)p(y)}{p(x)} dy \\ &= \frac{\gamma y^* l(x) \int_{-\infty}^{y^*} p(y) dx}{y l(x) + (1 - \gamma) g(x)} \\ &\propto (\gamma + \frac{g(x)}{l(x)} (1 - \gamma))^{-1} \end{aligned}$$

where

$$\gamma = p(y^* < y)$$

$$\begin{aligned} p(x|y) &= l(x) \text{ if } y < y^* \\ &= g(x) \text{ if } y \geq y^* \end{aligned}$$

and $p(a|b)$ is the conditional probability of a given event b .

The ranges of hyperparameters were devised to include all ranges explored by the various reports reviewed in the above section of this paper, as well as any other well-defined range or setting used in the field, yielding an immense search space with trillions of possible combinations. The hyperparameters included in the search space are listed in Table 1. Due to constraints in the Python package used for hyperparameter optimization (i.e., hyperas from hyperopt), a subsequent tuning of the window size, stride length and number of layers needed to be performed on the highest performing combination of all other hyperparameters via randomized grid search. This step was omitted in the current proof

of concept experiment, but the code for carrying out the grid search can be found in the file `accelstm/src/models/more_opt.py`. Thus, for initial optimization and the final cross validation (detailed below), data was partitioned using a window size of 128 with 50% stride length and fed into a 2-layer LSTM network.

For the second portion of the experiment, the Pipeline is completed via 5-fold cross validation, where the folds were made at the participant level so that no single participant's data ended up in both training and testing sets.

Languages and Libraries All models were written in the Python programming language. The LSTMs were built and run using the Keras library and TensorFlow as the backend heavy lifter. Hyperas from Hyperopt was used to optimize the network. Scikit learn provided the packages for cross validation, randomized grid search, and standardization of data. Numpy and Pandas were used to read and reformat the data among various other operations.

Results

During preliminary testing, we found that the model performed better on the total raw accelerometer signal (T) compared to the body-only data with the gravity-component (B) removed. As such, we used the total accelerometer signal (T) in our experiment.

The hyperparameter optimization explored a search space with trillions of possible parameter combinations. Due to time constraints, we stopped the search after six full days (hundreds of training iterations), during which time the suggested models' accuracies on test sets had ranged from 12.66% to 94.96%. The algorithm found several high-performing models and had used at least once all the values possible for each activation function, initialization strategy, regularization strategy, learning rate, and optimizer in the search space. The algorithm had tested models that both used and omitted batch normalization and bias, and it had tested dropout values between 0.005 and 0.991, batch sizes between 35 and 441 samples per batch, and from 10 to 508 units at both of the two layers.

Due to limited time to run our experiments, we conducted part two of the experiment concurrently with part one using a baseline LSTM architecture we felt would be a good starting point based on notes throughout the literature. The hyperparameter settings used in the model are as follows: window size, 128 time steps; step size, 50% of window size; number of layers, 2; units (layer1), 128; units (layer2), 114; batch size, 64; cell activation, tanh; recurrent activation, sigmoid; dropout, 0.5; weight initialization, Glorot Uniform; regularization, None; optimizer, RMSProp; bias, yes. We ran 5-fold CV on the model and computed the overall and class-wise F1 scores and accuracies. Cross validation yielded an average accuracy of 90.97% and F1 score of 0.90968, with a single best run of 95.25% accuracy and 0.9572 F1 score. We include the single best run for comparison with other reports, many of which do not report evidence of using cross validation or repeated trials.

Discussion

The execution of HAR research in various settings from the biomedical clinic early on [BMT⁺01], [RDML05], [BTvHS98] to current-day innovative settings such as the automobile [CFF⁺17], the bedroom [MAR⁺16], the dining room [KDD17], and outdoor sporting environments [REBS17] justifies the time spent expanding this area of research. As LSTM models are increasingly demonstrated to have potential for HAR research, the importance of deliberate and reproducible works is paramount.

Review of Previous Works A survey of the literature revealed a lack of cohesiveness regarding the use of LSTMs for accelerometer data and the overall data analysis pipeline. We grew concerned with possible sources of data leakage. Test set data should come from different participants than those used for the training data [HTF17], and no information from the test set should be exposed to the model before training.

We were surprised to see some of the more advanced preprocessing techniques being employed. Much of the appeal of non-linear models such as neural networks is their ability to learn from raw data itself and independently perform smoothing and feature extraction on noisy data through parameterized embeddings of the data. For example, Karpathy's 2015 study of LSTMs for language modeling showed specific neurons being activated when quotes were opened and deactivated when the quotes were closed, while others were activated by parenthetical phrases, marked the end of sentences, and so on [KJFF15]. Additionally, these preprocessing methods are more computationally expensive and less realistic for online and on-device implementations than is desired. The improved performance of the model on the total accelerometer signal (T) versus the body-only signal (B) with the gravity component removed demonstrates the promising potential of non-linear data-dependent models for classifying complex noisy data and supports our claim that extensive preprocessing is not necessary.

We do feel standardization is justified for this data due to its complexity and poor signal-to-noise ratio. Standardization is often important for data-dependent models such as LSTMs since the presence of outliers and skewed distributions may distort the weight embeddings [JWHT17].

Hyperparameter Optimization and Data Analysis Pipeline We structured our experiments with the objective of maintaining simplicity, relying as much as possible on the baseline model itself, maximizing generalizability and reproducibility of our methods and results, and unifying the existing methods and results in literature.

We saw very promising results from the hyperparameter optimization portion of the experiment. The TPE algorithm, although not run to completion in this experiment, was able to navigate the search space and find several well-performing models. We chose to err on the side of caution by using very granular ranges over the numerical hyperparameters, and as a result we ran out of time even using the heuristic-based TPE algorithm. We suggest further experiments to reduce the search space by using less granular ranges over the numeric hyperparameters, and exploring more advanced heuristic search methods. Doing so will decrease the search time and allow completion of the entire Pipeline in a more reasonable amount of time. Nonetheless, the TPE's so-far-best model at the time of termination and our baseline model from Part 2 outperformed other baseline LSTMs trained on higher dimensional data from the same dataset [U18], [ZYCG17]; see Table 2.

We also compare our performance with other benchmark experiments on the UCI HAR dataset. Compared with more complex LSTMs trained using more features, our averaged cross validation results scored competitively with the b-LSTM (91.09%), the residual LSTM (91.55%), and the deep res-bidir-LSTM (93.57%) all from Zhao, et. al. [ZYCG17]. As we found no evidence of cross validation in these other reports, we compare our single best-performing test's accuracy of 95.25% and F1 score of 0.9572 and find it to compete with the highest scoring models found in literature: 4 layer LSTM (96.7% accuracy, 0.96 F1score)

Category	Hyperparameter	Range
Data Processing	Window Size	24, 48, 64, 128, 192, 256
	Stride	25%, 50%, 75%
	Batch Size	32, 64, 128, ..., 480
Architecture	Units	2, 22, 42, 62, ..., 522
	Layers	1, 2, 3
Forward Processing	Activation Function (unit, state)	softmax, tanh, sigmoid, ReLU, linear
	Bias	True, False
	Weight Initialization (cell, state)	zeros, ones, random uniform dist., random normal dist., constant (0.1), orthogonal, Lecun normal, Glorot uniform
	Regularization (cell, state, bias, activation)	None, L2 Norm, L1 Norm
Regularization	Weight Dropout (unit, state)	uniform distribution (0, 1)
	Batch normalization	True, False
	Optimizers	SGD, RMSProp, Adagrad, Adadelata, Nadam, Adam
Learning	Learning Rate	$10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$

TABLE 1: The various hyperparameters included in the search space, and their respective ranges.

Model	Performance	Features
Baseline LSTM 1	90.77%	9 (T,B,G)
Baseline LSTM 2	85.35%	3-9 (?)
Pipeline P1 (Best)	93.47%	3
Pipeline P2 (CV)	90.97%	3
	0.90968	3
Pipeline P2 (Best)	95.25%	3
	0.9572	3

TABLE 2: Results table including results from baseline LSTM models trained on all 9 features provided in the dataset - total accelerometer signals (T), body accelerometer signals (gravity component removed, B), gyroscope signals (G). One of the baseline LSTM's did not explicitly specify the number of features used but only mentioned accelerometer signals. We provide results from Part 1 (P1, Hyperparameter Optimization) and Part 2 (P2, Cross-Validation) of our Pipeline. P2 scores include accuracies as percentages and F1 scores as decimals.

[MP17], MLSTM-FCN and MALSTM-FCN (96.71% accuracy) [KMDH18], and one-vs-one (OVO) SVM (96.4% accuracy, 551 features) [ROGA+13].

Conclusion/Future Work

We demonstrate the ability for a baseline LSTM model trained solely on raw triaxial accelerometer data (without gravity component removed) to perform competitively with classical models trained on hundreds of hand-crafted features and with other more complex LSTM models trained on higher dimensional sensor data.

We demonstrate the ability to optimize a data-centric model over an expansive hyperparameter search space and train it end-to-end within a scientifically rigorous and deliberate Data Analysis Pipeline. The code used in this project can be found at <https://github.com/xtianmcd/accelstm>.

Going forward, we would like to repeat this experiment to average performances from different models returned by the TPE algorithm; we would also like to repeat this experiment on other HAR datasets. Further exploration should be done to analyze why the algorithm's selections are indeed superior, how different data affect these choices, and how the LSTM cells within the models themselves are representing this type of data as has been done with LSTMs in other domains.

We hope that this Pipeline will serve useful in producing explicit and reproducible experiment results and in pushing the field forward in a methodical way.

REFERENCES

- [AGO+13] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. *21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013*, 2013.
- [BBBK11] James Bergstra, Remi Bardenet, Yoshua Bengio, and Balazs Kegl. Algorithms for hyper-parameter optimization. *NIPS*, 2011.
- [Ber17] Victor Bergelin. Human activity recognition and behavioral prediction using wearable sensors and deep learning. Master's thesis, Linkopings Universitet Matematiska Institutionen, 2017.
- [BI04] Ling Bao and Stephen S Intille. Activity recognition from user-annotated acceleration data. *Springer-Verlag*, 2004.
- [BMT+01] JB Bussmann, WL Martens, JH Tulen, FC Shasfoort, HJ van den Berg-Emons, and HJ Stam. Measuring daily behavior using ambulatory accelerometry - the activity monitor. *Behavior Research Methods, Instruments, and Computers*, 2001.
- [Bro17] Sofia Broome. Objectively recognizing human activity in body-worn sensor data with more or less deep neural networks. Master's thesis, KTH Royal Institute of Technology School of Computer Science and Communication, 2017.
- [BTvHS98] JB Bussmann, JH Tulen, EC van Herel, and HJ Stam. Quantification of physical activities by means of ambulatory accelerometry - a validation study. *Psychophysiology*, 1998.
- [CFF+17] Eduardo Carvalho, Bruno V Ferreira, Jair Ferreira, Cleidson de Souza, Hanna V Carvalho, Yoshihiko Suhara, Alex Sandy Pentland, and Gustavo Pessin. Exploiting the use of recurrent neural networks for driver behavior profiling. *2017 International Joint Conference on Neural Networks - IJCNN*, 2017.

- [CZZZ16] Yuwen Chen, Kunhua Zhong, Ju Zhang, and Xueliang Zhao. Lstm networks for mobile human activity recognition. *2016 International Conference on Artificial Intelligence*, 2016.
- [FFH⁺16] Madalina Fiterau, Jason Fries, Eni Halilaj, Nopphon Siranart, Suvrat Bhooshan, and Christopher Re. Similarity-based lstms for time series representation learning in the presence of structured covariates. *29th Conference on Neural Information Processing Systems - NIPS 2016*, 2016.
- [GBGG16] Hristijan Gjoreski, Jani Bizjak, Martin Gjoreski, and Marjaz Gams. Comparing deep and classical machine learning methods for human activity recognition using wrist accelerometer. Technical report, Jozef Stefan Institute Department of Intelligent Systems, 2016.
- [Ger17] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. OReilly Media, Inc., Sebastopol, CA, 2017.
- [GP17] Yu Guan and Thomas Plotz. Ensembles of deep lstm learners for activity recognition using wearables. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2017.
- [GRX16] Weixuan Gao, Chuanwei Ruan, and Rui Xu. Sensor-based semantic-level human activity recognition using temporal classification. Technical report, Stanford University, 2016.
- [GSS02] Felix A Gers, Nicol N Schraudolph, and Jurgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research*, 2002.
- [HDJS18] B Hu, PC Dixon, JV Jacobs, and JM Schiffman. Machine learning algorithms based on signals from a single wearable inertial sensor can detect surface- and age-related differences in walking. *Journal of Biomechanics*, 2018.
- [HHP16] Nils Y Hammerla, Shane Holloran, and Thomas Plotz. Deep, convolutional, and recurrent models for human activity recognition using wearables. *ArXiv*, 2016.
- [HS97] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.
- [HTF17] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning*. Springer, 2017.
- [JWHT17] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, New York, 2017.
- [JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. *32nd International Conference on Machine Learning*, 2015.
- [KDD17] Konstantinos Kyriasis, Christos Diou, and Anastasios Delopoulos. Food intake detection from inertial sensors using lstm networks. *New Trends in Image Analysis and Processing - ICIAP*, 2017.
- [KHC10] Eunju Kim, Sumi Helal, and Diane Cook. Human activity recognition and pattern discovery. *IEEE Persuasive Computing*, 9(1), 2010.
- [KJFF15] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *ArXiv*, 2015.
- [KMDH18] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Samuel Harford. Multivariate lstm-fns for time series classification. *ArXiv*, 2018.
- [LBMG15] Gregoire Lefebvre, Samuel Berlemont, Franck Mamalet, and Christophe Garcia. *Inertial Gesture Recognition with BLSTM-RNN*. Springer Series in Bio-/Neuroinformatics 4 - Artificial Neural Networks, Switzerland, 2015.
- [LC12] Myeong-Chun Lee and Sung-Bae Cho. Mobile gesture recognition using hierarchical recurrent neural network with bidirectional long short-term memory. *6th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2012.
- [MAR⁺16] Arnaud Moreau, Peter Anderer, Marco Ross, Andreas Cerny, Timothy Almazan, and Barry Peterson. Detection of nocturnal scratching movements in patients with atopic dermatitis using accelerometers and recurrent neural networks. *IEEE Journal of Biomedical and Health Informatics*, 2016.
- [Mil18] John A Miller. Introduction to data science using scalation. Technical report, University of Georgia Department of Computing Science, 2018.
- [MKS17] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *ArXiv*, 2017.
- [MMB⁺18] Mirto Musci, Daniele De Martini, Nicola Blago, Tullio Facchinetti, and Marco Piastra. Online fall detection using recurrent neural networks. *ArXiv*, 2018.
- [MP43] WS McCulloch and W Pitts. A logical calculus of the ideas immanent in neurons activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.
- [MP17] Abdulmajid Murad and Jae-Young Pyun. Deep recurrent neural networks for human activity recognition. *Sensors*, 17(11), 2017.
- [MS10] Andrea Mannini and Angelo Maria Sabatini. Machine learning methods for classifying human physical activity from on-body accelerometers. *Sensors*, 2010.
- [OR16] Javier Ordonez and Daniel Roggen. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 2016.
- [PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ArXiv*, 2013.
- [PW17] Ofir Press and Lior Wolf. Using the output embedding to improve language models. *ArXiv*, 2017.
- [RDML05] Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael L Littman. Activity recognition from accelerometer data. *IAAI 2005 Proceedings of the 17th conference on Innovative applications of artificial intelligence*, 3:1541–1546, 2005.
- [REBS17] Aliaa Rassem, Mohammed El-Beltagy, and Mohamed Saleh. Cross-country skiing gears classification using deep learning. *ArXiv*, 2017.
- [RG17] Nils Reimers and Iryna Gurevych. Optimal hyperparameters for deep lstm-networks for sequence labeling tasks. *ArXiv*, 2017.
- [ROGA⁺13] Jorge Luis Reyes-Ortiz, Alessandro Ghio, Davide Anguita, Xavier Parra, Joan Cabestany, and Andreu Catala. Human activity and motion disorder recognition - towards smarter interactive cognitive environments. *21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013*, 2013.
- [Ros57] Frank Rosenblatt. The perceptron - a perceiving and recognizing automation. Technical report, Cornell Aeronautical Laboratory, 1957.
- [RVCK17] Patricio Rivera, Edwin Valerezo, Mun-Taik Choi, and Tae-Seong Kim. Recognition of human hand activities based on a single wrist imu using recurrent neural networks. *International Journal of Pharma Medicine and Biological Sciences*, 2017.
- [Set18] Dan Setterquist. Using a smartphone to detect the standing-to-kneeling and kneeling- to-standing postural transitions. Master's thesis, KTH Royal Institute of Technology School of Electrical Engineering and Computer Science, 2018.
- [SKP18] W Seok, Y Kim, and C Park. Pattern recognition of human arm movement using deep reinforcement learning. *2018 International Conference on Information Networking - ICOIN*, 2018.
- [SS16] Sungho Shin and Wonyong Sung. Dynamic hand gesture recognition for wearable devices with low complexity recurrent neural networks. *2016 IEEE International Symposium on Circuits and Systems - ISCAS*, 2016.
- [SS17] Henrik Sjostrum and Carlos Nieves Sanchez. Deepconvlstm on single accelerometer locomotion recognition. Technical report, UMEA Universitet, 2017.
- [U18] Chandini U. A machine learning based activity recognition for ambient assisted living. *International Journal on Future Revolution in Computer Science and Communication Engineering*, 4(3), 2018.
- [WA17] Elias Wu and Paa Adu. What am i doing - robust human activity detection with smartphones athletics and sensing devices. Technical report, Stanford University, 2017.
- [WRSN14] Simon Wiesler, Alexander Richard, Ralf Schluter, and Hermann Ney. Mean-normalized stochastic gradient for large-scale deep learning. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2014.
- [WZ89] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1989.
- [ZSO17] Tahmina Zebin, Patricia J Scully, and Krikor B Ozanyan. Human activity recognition with inertial sensors using a deep learning approach. *Sensors*, 2016 IEEE, 2017.
- [ZWYM16] Rui Zhao, Jinjiang Wang, Ruqiang Yan, and Kezhi Mao. Machine health monitoring with lstm networks. *2016 10th International Conference on Sensing Technology - ICST*, 2016.
- [ZYCG17] Yu Zhao, Rennong Yang, Guillaume Chevalier, and Maoguo Gong. Deep residual bidir-lstm for human activity recognition using wearable sensors. *ArXiv*, 2017.
- [ZYH⁺18] Xiang Zhang, Lina Yao, Chaoran Huang, Sen Wang, Mingkui Tan, Guodong Long, and Can Wang. Multi-modality sensor data classification with selective attention. *2018 International Joint*

*Conference on Artificial Intelligence At Stockholm, Sweden,
2018.*

Practical Applications of Astropy

David Shupe^{‡*}, Frank Masci[‡], Russ Laher[‡], Ben Rusholme[‡], Lee Armus[‡]

<https://youtu.be/2GTLkH5sfJc>

Abstract—Packages developed under the auspices of the Astropy Project ([ART+13], [TPS+18]) address many common problems faced by astronomers in their computational projects. In this paper we describe how capabilities provided by Astropy have been employed in two current projects. The data system for the Zwicky Transient Facility processes a terabyte of image data every night, with a lights-out automated pipeline that produces difference images about ten minutes after the receipt of every exposure. Astropy is used extensively in the astrometry and light-curve-generation modules, making especially heavy use of FITS header manipulation, table I/O, and coordinate conversion and matching. The second project is a web application made with Plotly Dash for proposal studies for the Origins Space Telescope. The `astropy.cosmology` module provided easy redshifting of our template galaxy spectrum, and `astropy.units` enabled the porting of an instrument sensitivity function to Python, with verification that a very complex combination of units resulted in a dimensionless signal-to-noise value.

Index Terms—astronomy, data processing

Introduction

The Astropy Project is a community-driven effort to provide both a core Python package of functionality commonly used by astronomers, and an extended ecosystem of interoperable packages, with high standards for documentation and testing ([ART+13], [TPS+18]). The astropy core package includes subpackages for representing and manipulating space and time coordinates; I/O for astronomical file formats; world coordinate systems in images (e.g. converting between celestial coordinates and image pixels); cosmological calculations; and manipulating numerical quantities with units. Most astronomers using the astropy core package use it for interactive analyses. In this paper, we highlight the importance of astropy in two production environments: the data system for the Zwicky Transient Facility (ZTF), and a web application for the proposed Origins Space Telescope.

The ZTF Project

The Zwicky Transient Facility (ZTF) is a new robotic survey now underway, using the 48-inch Samuel Oschin Telescope at Palomar Observatory in southern California. This telescope was originally constructed to take images with photographic plates, in large part to provide targets for the 200-inch Hale Telescope at the same observatory. The ZTF camera fills the focal plane of the 48-inch

telescope with sixteen $6k \times 6k$ charge-coupled devices (CCDs) with an active detector area of 47 square degrees (Dekany et al in prep; [DSB+16]). ZTF is conducting a fast, wide-area time-domain survey (Bellm et al in prep) designed to discover fast, young and rare flux transients; counterparts to gravitational wave sources; low-redshift Type Ia supernovae for cosmology; variable stars and eclipsing binaries; and moving objects in our Solar System such as asteroids (Graham et al in prep). The entire sky visible to Palomar can be imaged each night to declinations above -30 degrees. The survey began in March 2018 and will continue for three years. Figure 1 shows a field-of-view comparison of ZTF with its predecessor at Palomar, the Palomar Transient Factory (PTF; [LKD+09]), and the forthcoming Large Synoptic Survey Telescope (LSST).

A typical night of ZTF observations includes about 750 exposures totaling about 1 Terabyte of image data when uncompressed. Each quadrant of the CCDs is processed separately for a total of about 55,000 calibrated science images per night. Depending on sky location, 0.5 to 1 billion individual source measurements are extracted per night. The ZTF data system (Masci et al. 2018, in review, [LMG+18]) is operated by the IPAC data center on the Caltech campus. Within a few minutes of receipt of an exposure at IPAC, a real-time image subtraction pipeline outputs alert packets of potential transient objects, at rates already nearing 1 million per night. Alert packets from the public portion of the survey have just become available¹, along with a repository of the schema and code examples².

The data system is mostly scripted in Perl, with job management relying on a Postgres database. A cluster of 66 compute nodes handles the processing. Astropy is used in several key components of the pipeline. In the following subsections we outline Astropy use and what we've learned from operational experience.

Improving reliability of the astrometric solver

Assigning coordinates to ZTF images is challenging for several reasons. The accuracy of the pointing of the boresight (center of the field-of-view) is about 20 arcseconds rms. Atmospheric effects cause image distortions on small scales, and these effects are exacerbated at low elevations. ZTF employs the *Scamp* astrometric solver from the Astromatics suite ([Ber06]) to match star positions from the Gaia Data Release 1 (DR1) catalog ([GPd+16], [GBV+16]) and ultimately fit a 4th-order polynomial to the image distortions. *Scamp* is written in C and requires inputs in a very

* Corresponding author: shupe@ipac.caltech.edu
[‡] Caltech/IPAC

Copyright © 2018 David Shupe et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. <https://ztf.uw.edu/alerts/public>

2. <https://github.com/ZwickyTransientFacility/ztf-avro-alert>

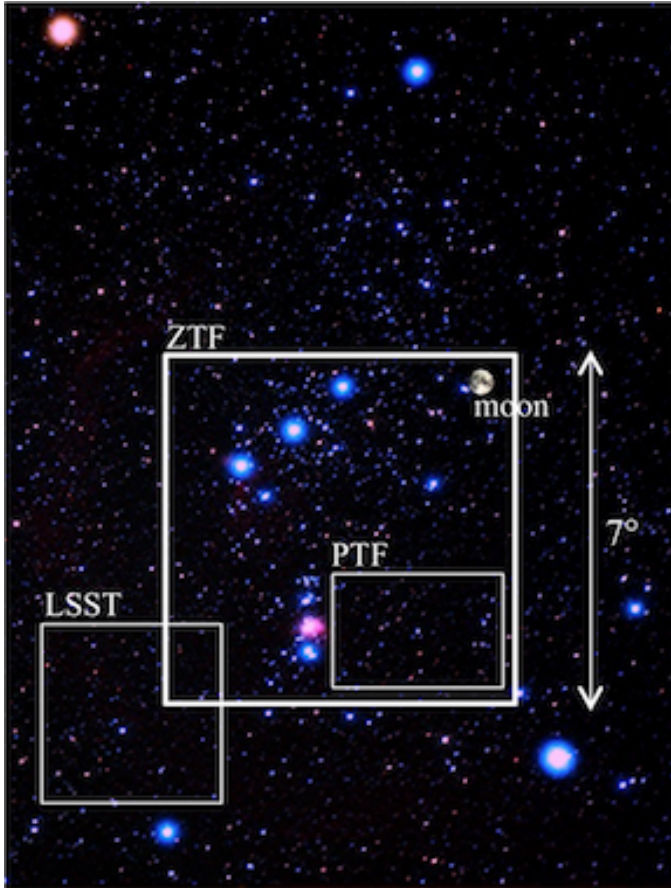


Fig. 1: Field of view of the ZTF camera, compared to the predecessor Palomar Transient Factory (PTF) camera, and the forthcoming Large Synoptic Survey Telescope (LSST). The background image shows the Orion constellation.

specialized format. We have developed a procedure that has significantly reduced the rate of incorrect solutions in crowded fields, by providing *Scamp* with an accurate starting point (see Figure 2).

Scamp requires both the input catalog of detections and the reference catalog to be provided in LDAC (Leiden Data Analysis Center)³ FITS format. This format consists of header information encoded in a binary format in a table extension, followed by another table extension of detections. Recent versions of *Scamp* will start from a prior World Coordinate System (WCS; [CG02]) solution provided to the program. Providing a distortion prior derived from many observations makes it much easier for *Scamp* to converge on the global minimum, i.e. the correct distortion solution. Our efforts to include the WCS in the LDAC file of detections using *astropy.io.fits* were unsuccessful. However, the WCS information in the LDAC file can be overridden by a text file of header information provided separately to *Scamp*.

Our distortion prior is constructed from an offline analysis of images taken at high elevations (low airmasses), the same conditions used in the ZTF survey. For selected fields, we set up idealized WCS objects with 1 degree per "pixel":

```
from astropy.wcs import WCS
field_radec = {619: (143.619, 26.15),
               620: (151.101, 26.15),
```

3. https://marvinweb.astro.uni-bonn.de/data_products/THELIWWW/LDAC/LDAC_concepts.html

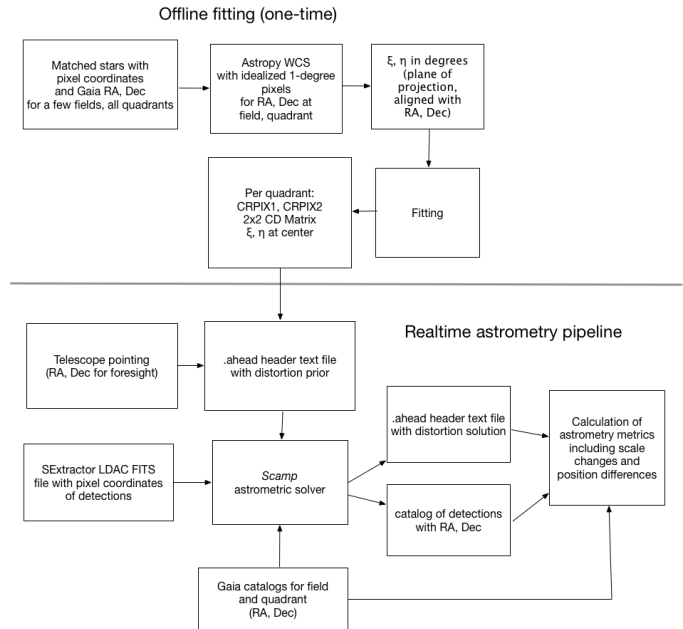


Fig. 2: Processing diagrams for ZTF astrometry. An offline analysis (top) is performed on a few exposures to make a prior model for each of the 64 quadrants in the focal plane. These terms are combined with telescope pointing to make an accurate prior for *Scamp* in the realtime pipeline (bottom), resulting in a calibrated header text file with the full coordinate solution including distortions, and a catalog of the detected stars with assigned RA and Dec coordinates. These outputs of the astrometric fitting are matched again with the Gaia reference catalog to produce metrics for assessing the quality of the astrometric fit.

```
665: (133.35, 33.35),
667: (149.057, 33.35) }
```

```
wdict = {}
for field, (ra, dec) in field_radec.items():
    w = WCS(naxis=2)
    w.wcs.crpix = [0.0, 0.0]
    w.wcs.cdelt = np.array([1.0, 1.0])
    w.wcs.crval = [ra, dec]
    w.wcs.ctype = ["RA---TAN", "DEC--TAN"]
    wdict[field] = w
```

Then when reading in a catalog of sources with positions for each field, we convert the right ascensions and declinations to projection plane coordinates ([CG02]) ξ , η in units of degrees in the tangent plane:

```
w = wdict[field]
plane_coords = w.wcs_world2pix(
    np.vstack([tab['ra'], tab['dec']]).T, 1)
xi = plane_coords[:, 0]
eta = plane_coords[:, 1]
```

A linear model is fit relating image pixel values to the computed ξ and η values, while allowing offsets and linear terms for each exposure and readout channel. This fit yields the CRPIX1 and CRPIX2 values (pixel offsets) from the telescope boresight to each of the 64 readout channels. This linear solution yields residuals of about four arcseconds in magnitude. Then "global" pixel coordinates are constructed and a quadratic fit relating these to ξ and η is computed. This second fit is used to find ξ and η for the center of each quadrant-image. For each quadrant-image, a linear fit is made to yield the multiplicative terms for pixel scale and rotation (CD-matrix values; [CG02]) for each quadrant. This

procedure transfers the pointing to the center of each individual quadrant-image.

The CD-matrix, CRPIX1, CRPIX2, and ξ , η values for each quadrant are saved to be used by the astrometry pipeline. The parameters are read and inserted into a text file (.ahead file) that initializes *Scamp*. For each image, a first run of *Scamp* is made using 'PRE-DISTORTED' mode. This performs pattern-matching of detected stars and reference stars from Gaia DR1. *Scamp* is allowed only a little freedom to rotate and change scale. A second pass of *Scamp* skips the pattern-matching and fits a fourth-degree distortion polynomial as part of the output WCS. An essential speed improvement was pre-fetching static copies of the Gaia DR1 catalog and storing these in the LDAC FITS format using `astropy.io.fits`, in a static area, to be available as static catalogs for *Scamp*.

Assessing the quality of the astrometric solution

A problem encountered often in the PTF survey was not being able to readily tell whether a solution output by *Scamp* was of poor quality. Astrometric problems greatly increase the number of spurious transients produced by image subtraction and later steps of the pipeline and scanning processes. *Scamp* does output a chi-squared statistic. When provided with realistic errors, most good solutions result in a chi-squared statistic of about five. To ensure that the system catches the case of a distortion polynomial that is unconstrained in the corner of an image, we developed a scale check test of the final solution against the distortion prior that we initially provided to *Scamp*.

First we generate a grid over the detector, and then make pixel coordinates at each grid point:

```
y_pix1 = np.arange(1, 3082, 140)
x_pix1 = np.arange(1, 3074, 128)
m1 = np.meshgrid(x_pix1, y_pix1)
mx = np.array(m1[0].flat)
my = np.array(m1[1].flat)
pcoords = np.vstack([mx,my]).T
pcoordsb = np.vstack([mx+1,my+1]).T
pcoordsr = np.vstack([mx+1,my]).T
pcoordsd = np.vstack([mx,my+1]).T
```

Then using the WCS objects from the prior and from the final solution, we calculate pixel areas:

```
from astropy.coordinates import SkyCoord
import astropy.units as u
finalcoords = SkyCoord(wcs_final.all_pix2world(
    pcoords, 1), unit=u.deg, frame='icrs')
finalcoordsb = SkyCoord(wcs_final.all_pix2world(
    pcoordsb, 1), unit=u.deg, frame='icrs')
finalcoordsr = SkyCoord(wcs_final.all_pix2world(
    pcoordsr, 1), unit=u.deg, frame='icrs')
finalcoordsd = SkyCoord(wcs_final.all_pix2world(
    pcoordsd, 1), unit=u.deg, frame='icrs')
finalareas = (finalcoords.separation(finalcoordsb)*
    finalcoordsr.separation(finalcoordsd))/2
```

These steps are repeated for the prior. Finally we compute a percentage change in pixel scale:

```
pctscaledif = 100*(np.sqrt(finalareas) -
    np.sqrt(priorareas))/np.sqrt(priorareas)
```

If the percentage scale difference changes by more than a percent, the image is marked as unusable. Figure 3 shows the mean value of the percentage scale difference for a night of ZTF commissioning exposures, showing the changes follow a model⁴ for differential atmospheric refraction.

4. <http://wise-obs.tau.ac.il/~eran/Wise/Util/Refraction.html>

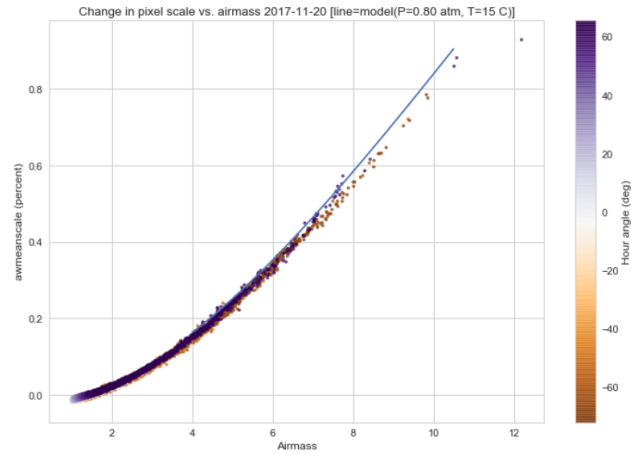


Fig. 3: Mean pixel scale versus airmass for one night of commissioning data. The line shows model points for pressure and temperature appropriate for Palomar Observatory.

A peculiarity for ZTF is that with a field-of-view that is seven degrees on a side, the airmass reported by the telescope control system does not apply well for the outer CCDs. We use an AltAz model to recompute airmass when analyzing metric values for the pixel scale change.

```
palomar = EarthLocation.of_site('palomar')
time = Time(df.obs_mjd, format='mjd')
coords = SkyCoord(ra=df.ra0, dec=df.dec0,
    unit=u.deg, frame='icrs',
    obstime=time,
    location=palomar)
altaz = coords.transform_to(
    AltAz(obstime=time,
    location=palomar))
df['secz'] = altaz.secz
```

A future update to the astrometry module, now being tested, distorts the CD-matrix along the azimuthal direction and by a magnitude determined from the differential refraction model. The correction is not needed for the main survey and will help find solutions for targets of opportunity at high airmass.

Accounting for light-travel-time in ZTF light curves

For ZTF, the PSF-fitting photometry that is extracted from every image is periodically combined into matchfiles in HDF5 format. These matchfiles form the basis of the lightcurve service that will be deployed by IPAC's Infrared Science Archive. The matchfiles are also used to provide light curves for variable star studies.

The matchfiles are seeded by PSF-fitting photometry extracted from reference images. The reference images are coadds of between 15 and 40 exposures of a ZTF field. Astropy's `SkyCoord` class is employed to perform the matching of input sources to reference objects.

Astropy is also used to provide heliocentric Julian dates for each source. The difference between heliocentric Julian date and observed Julian date is the light-travel time difference between the Earth-to-coordinate direction and the Sun-to-coordinate direction. It is computationally prohibitive to compute this time difference for each individual source. Instead, a `SkyOffset` frame is defined at the maximum coordinate for a field, and then a 9x9 grid is set up on that offset grid. A fit is made of light-travel-time difference as a quadratic function of longitude and latitude in the offset frame.

This provides an accuracy in the calculation of the heliocentric date that is much less than a ZTF exposure time of 30 seconds.

Since some ZTF fields straddle RA=0, a mean or median of RA yields misleading values. For our nearly-degree-sized fields, we use the maximum values and define an offset frame:

```
import numpy as np
from astropy.coordinates import SkyCoord
import units as u

max_ra = np.max(ra)
max_dec = np.max(dec)
# Make calculations in sky offset frame
max_coord = SkyCoord(ra=max_ra*u.deg,
                    dec=max_dec*u.deg)
aframe = max_coord.skyoffset_frame()
```

The PSF-fitting catalog coordinates are transformed to the offset frame and a bounding box in that frame is computed:

```
psfcoords = SkyCoord(ra=ra*u.deg,
                    dec=dec*u.deg)
psfcoords = psfcoords.transform_to(aframe)
min_lon = np.min(psfcoords.lon)
max_lon = np.max(psfcoords.lon)
min_lat = np.min(psfcoords.lat)
max_lat = np.max(psfcoords.lat)
```

A 9x9 grid is set up in the SkyOffset frame:

```
grid_lon = np.linspace(min_lon.value,
                      max_lon.value,
                      endpoint=True,
                      num=9)
grid_lat = np.linspace(min_lat.value,
                      max_lat.value,
                      endpoint=True,
                      num=9)
glon, glat = np.meshgrid(grid_lon, grid_lat)
glon, glat = glon.flatten(), glat.flatten()
gcoords = SkyCoord(lon=glon*u.deg,
                  lat=glat*u.deg, frame=aframe)
```

Although `coord.EarthLocation.of_site` was used in our offline astrometry analysis, its network fetch of coordinates is not reliable for many parallel processes. The hard-coded observatory location is combined with the modified Julian date of the observation to compute light-travel-time over our 9x9 grid:

```
from astropy import time

palomar = coord.EarthLocation.from_geocentric(
    -2410346.78217658,
    -4758666.82504051,
    3487942.97502457, u.m)
mytime = time.Time(mjd, format='mjd', scale='utc',
                  location=palomar)
ltt_helio = mytime.light_travel_time(gcoords,
                                    'heliocentric')
```

Coefficients for a least-squares fit of a 2-dimensional quadratic surface are computed and applied to our catalog coordinates to yield light-travel-times for each source, and then added to our observed times to result in heliocentric Julian dates:

```
A = np.c_[np.ones(glon.shape), glon, glat,
          glon*glat, glon**2, glat**2]
coeffs, _, _ = np.linalg.lstsq(A, ltt_helio.sec)
fitted = np.dot(np.c_[np.ones(psfcoords.lon.shape),
                    psfcoords.lon.value,
                    psfcoords.lat.value,
                    psfcoords.lon.value*psfcoords.lat.value,
                    psfcoords.lon.value**2,
                    psfcoords.lat.value**2],
               coeffs).reshape(psfcoords.lon.shape)
hjd = mytime + fitted*u.s
```

Configuration file issue

In the course of running the ZTF pipeline in production, we encountered a serious problem caused by the `$HOME/.astropy/config` file. This file would randomly corrupt, causing every Astropy import to fail. The cause of the problem was different Astropy versions installed in our Python 2 & 3 virtual environments. The configuration file is overwritten every time a different version of Astropy version is imported. Our pipeline contained a mixture of Python 2 and Python 3 code, running in parallel at enough scale, that a collision would eventually occur. The problem was solved by installing the same version of Astropy in both versions of Python.

Lessons learned from the ZTF experience

- Python and Astropy worked very well to wrap the *Scamp* solver and to provide its specialized inputs to make it converge reliably on correct astrometric solutions.
- The key to working with the LDAC format is providing an additional text file header that is easily manipulated with Astropy.
- Astropy.wcs supports TPV distortions since version 1.1, enabling us to compute metrics assessing the quality of the astrometric fits.
- When you have a 7-degree field of view, the elevation, azimuth, and airmass reported by the telescope system lack sufficient precision.
- Eliminate network calls as much as possible, by pre-fetching the astrometric catalogs, and bypassing `astropy.coordinates.EarthLocation.of_site`.
- `SkyCoord.offset_frame` is essential to avoid zero-wrapping problems in celestial coordinates, and is very useful when working on a patch of sky.
- Configuration files can cause problems at scale.
- Technical debt from not converting everything to Python 3 will bite you.

Origins Space Telescope

The Origins Space Telescope is a space observatory concept under study as part of NASA's astrophysics roadmap. The first design includes a 9-meter primary mirror with all components cooled to less than 6 K, to provide orders of magnitude more sensitivity than previous space infrared missions.

As part of the concept study, a web application has been constructed to showcase the potential of one of the spectroscopic instruments, the Mid-Resolution Survey Spectrometer ([BO18]). The purpose of the application is to allow trade studies of different observational parameters, including the telescope diameter, the exposure time, and the distance to the star or galaxy of interest. Plotly Dash⁵ was chosen as the technology for constructing the project.

Part of the project involved converting a complicated function for instrument sensitivity to Python. The `astropy.units` and `astropy.constants` packages made it relatively easy to check the results of the calculation.

Many astronomers are used to working with "magic numbers" that are constants or combinations of constants that we keep in our heads. Here is an example:

5. <https://plot.ly/products/dash/>

```
freq=double(2.9979e5/wave) ; in GHz
h=double(6.626e-18) ; h in erg / GHz
c=double(2.9979e10) ; c in cm / sec
```

With `astropy.units` and affiliated packages:

```
import astropy.constants as const
import astropy.units as u
```

```
freq = const.c/wave
```

The noise equivalent flux calculation for the spectrometer depends in part on the numbers of photons (occupation number) coming from the background at a particular wavelength.

$$\bar{n} = \frac{c^2 I_\nu}{2h\nu^3}$$

where I_ν is the background intensity in MJy/sr. An assertion in the calculation of occupation number ensures it is dimensionless:

```
def occnum_bkg(wave, background):
    """
    returns photon occupation
    number from background
    """

    freq=const.c/wave

    occnum = (u.sr*const.c**2*background/
              (2*const.h*freq**3)
              # background is provided in MJy / sr
              assert occnum.unit.is_equivalent(
                  u.dimensionless_unscaled)
              return occnum
```

The assertion ensures that the occupation number is dimensionless.

The noise equivalent power for an element in the spectrometer depends the frequency, bandwidth and photon occupation number at that frequency:

$$NEP = h\nu\sqrt{\Delta\nu\bar{n}(\bar{n}+1)}$$

where the bandwidth $\Delta\nu = \nu/R$ and R is the spectrometer resolution. In the instrument sensitivity function, this is implemented with an assertion to check units at an intermediate stage:

```
delta_freq = freq / resolution
nep_det = (const.h*freq*
           np.sqrt(delta_freq*nbar*(nbar+1))
           *sqrt(2)) # in W/sqrt(Hz)
assert nep_det.unit.is_equivalent(u.W*u.Hz**-0.5)
```

For the extragalactic example in the application, the `astropy.cosmology` module was used to redshift the spectrum. The Planck 2015 cosmology ([PAA⁺16]) is one of the built-in cosmologies in the package. For each user-selected value of redshift, we computed the luminosity distance to scale the flux values of the spectrum.

For re-gridding the wavelength spectrum, we used the `pysynphot` package (not an `astropy` package but developed in part by `Astropy` developers) ([STS13]) to interpolate the redshifted spectrum onto the observed wavelength channels.

The application has been deployed on the Heroku platform⁶. A screenshot of the galaxy spectrum is shown in Figure 4. To ensure good performance when changing parameters, the instrument sensitivity was pre-computed for the lines in the spectra, for different backgrounds and redshifts.

6. <https://ost-mrss.herokuapp.com>

The `astropy.units` package is broadly useful outside astronomy; to that end, the `unyt` package ([GZT⁺18]) is a newly-available standalone alternative.

Lessons learned include:

- Using a units package together with assertions at intermediate stages helped to validate a complex instrument sensitivity function.
- However, a units package does not help get factors of $(1+z)$ correct.
- Pre-computing sensitivities for several parameter choices sped up the application.
- The `pysynphot` functionality for re-gridding spectra would be useful to break out into a more accessible `Astropy`-affiliated package.

Conclusions

This paper highlights the use of `Astropy` in two production environments: the Zwicky Transient Facility data system, and a web application for the Origins Space Telescope. `Astropy`'s capabilities for manipulating FITS files and image headers, coupled with its coordinate conversion capabilities, helped us implement a scheme to greatly improve the reliability of ZTF astrometry, and provided other conveniences. The `astropy.units` and `astropy.cosmology` packages provided essential transformations for the Origins study application. We found that some care needs to be taken with minimizing or eliminating network calls, and with handling configuration files that assume a single package version is in use.

Acknowledgments

We are grateful to D. Levitan, W. Landry, S. Groom, B. Sesar, J. Surace, E. Bellm, A. Miller, S. Kulkarni, T. Prince and many other contributors to the PTF and ZTF projects. The Origins Space Telescope app includes significant contributions from C.M. Bradford, K. Pontopiddan, K. Larson, J. Marshall, and T. Diaz-Santos.

ZTF is led by the California Institute of Technology, US and includes IPAC, US; the Joint Space-Science Institute (via the University of Maryland, College Park), US; Oskar Klein Centre of the University of Stockholm, Sweden; University of Washington, US; Weizmann Institute of Science, Israel; DESY and Humboldt University of Berlin, Germany; University of Wisconsin at Milwaukee, US; the University System of Taiwan, Taiwan; and Los Alamos National Laboratory, US; ZTF acknowledges the generous support of the National Science Foundation under AST MSIP Grant No 1440341. The alert distribution service is provided by the DIRAC Institute at the University of Washington. The High Performance Wireless Research & Education Network (HP-WREN; <https://hpwren.ucsd.edu>) is a project at the University of California, San Diego and the National Science Foundation (grant numbers 0087344 (in 2000), 0426879 (in 2004), and 0944131 (in 2009)).

This work has made use of data from the European Space Agency (ESA) mission Gaia (<https://www.cosmos.esa.int/gaia>), processed by the Gaia Data Processing and Analysis Consortium (DPAC, <https://www.cosmos.esa.int/web/gaia/dpac/consortium>). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the Gaia Multilateral Agreement.

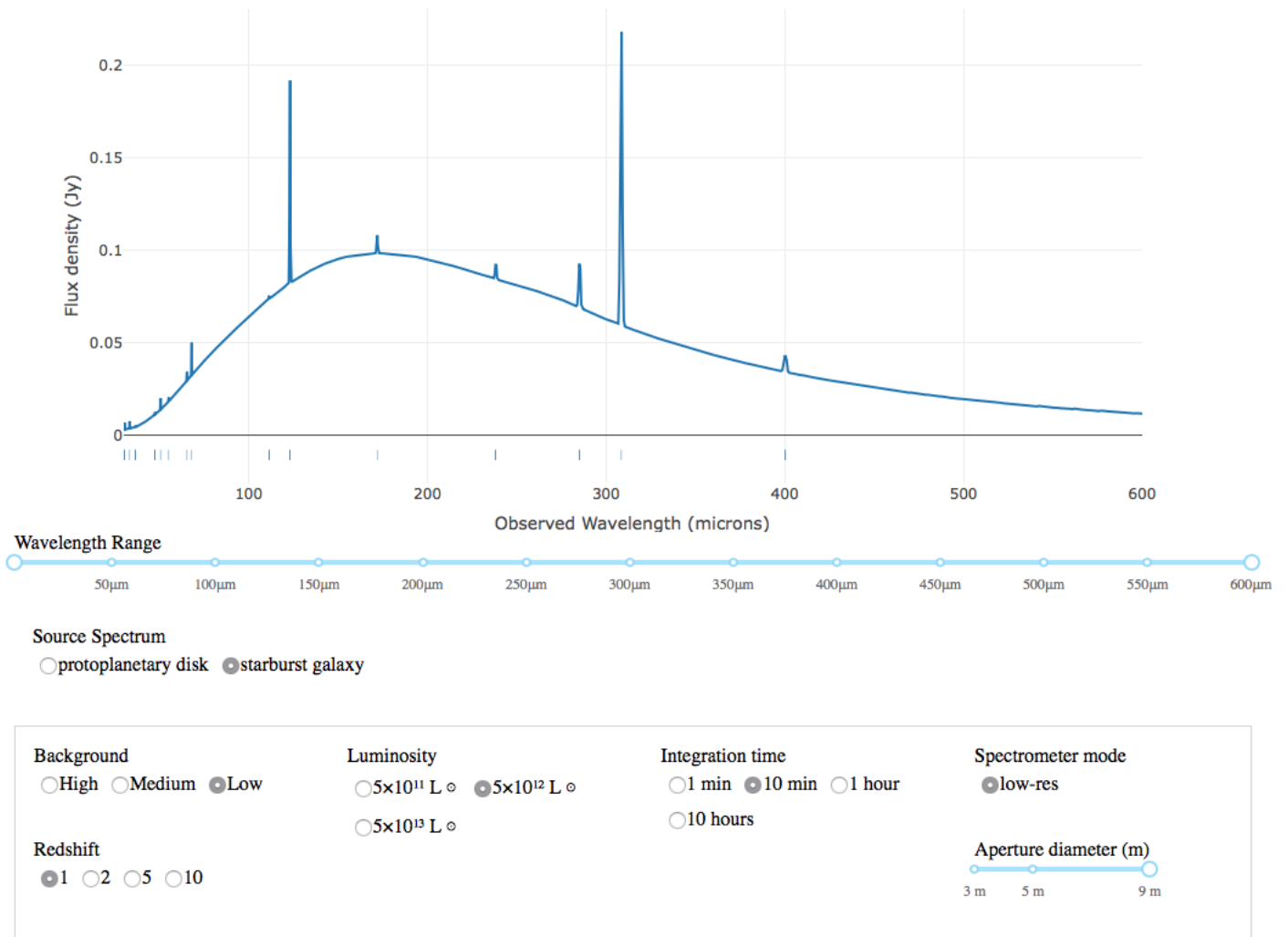


Fig. 4: The web application for the Origins Space Telescope, showing the galaxy spectrum and controls for changing source characteristics and instrument parameters.

REFERENCES

- [ART⁺13] Astropy Collaboration, Thomas P. Robitaille, Erik J. Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis, Adam Ginsburg, Adrian M. Price-Whelan, Wolfgang E. Kerzendorf, Alexander Conley, Neil Crighton, Kyle Barbary, Demitri Muna, Henry Ferguson, Frédéric Grollier, Madhura M. Parikh, Prasanth H. Nair, Hans M. Unther, Christoph Deil, Julien Woillez, Simon Conseil, Roban Kramer, James E. H. Turner, Leo Singer, Ryan Fox, Benjamin A. Weaver, Victor Zabalza, Zachary I. Edwards, K. Azalee Bostroem, D. J. Burke, Andrew R. Casey, Steven M. Crawford, Nadia Dencheva, Justin Ely, Tim Jenness, Kathleen Labrie, Pey Lian Lim, Francesco Pierfederici, Andrew Pontzen, Andy Ptak, Brian Refsdal, Mathieu Servillat, and Ole Streicher. Astropy: A community Python package for astronomy. *Astronomy and Astrophysics*, 558:A33, October 2013. doi:10.1051/0004-6361/201322068.
- [Ber06] E. Bertin. Automatic Astrometric and Photometric Calibration with SCAMP. In C. Gabriel, C. Arviset, D. Ponz, and S. Enrique, editors, *Astronomical Data Analysis Software and Systems XV*, volume 351 of *Astronomical Society of the Pacific Conference Series*, page 112, July 2006.
- [BO18] Charles Matt Bradford and Origins Space Telescope Study Team. The Medium Resolution Survey Spectrometer (MRSS) for the Origins Space Telescope: Enabling 3-D Surveys of the Universe in the Far-IR. In *American Astronomical Society Meeting Abstracts*, January 2018.
- [CG02] M. R. Calabretta and E. W. Greisen. Representations of celestial coordinates in FITS. *Astronomy and Astrophysics*, 395:1077–1122, December 2002. arXiv:astro-ph/0207413, doi:10.1051/0004-6361:20021327.
- [DSB⁺16] Richard Dekany, Roger M. Smith, Justin Belicki, Alexandre Delacroix, Gina Duggan, Michael Feeney, David Hale, Stephen Kaye, Jennifer Milburn, Patrick Murphy, Michael Porter, Daniel J. Reiley, Reed L. Riddle, Hector Rodriguez, and Eric C. Bellm. The Zwicky Transient Facility Camera. In *Ground-based and Airborne Instrumentation for Astronomy VI*, volume 9908, page 99085M, August 2016. doi:10.1117/12.2234558.
- [GBV⁺16] Gaia Collaboration, A. G. A. Brown, A. Vallenari, T. Prusti, J. H. J. de Bruijne, F. Mignard, R. Drimmel, C. Babusiaux, C. A. L. Bailer-Jones, U. Bastian, and et al. Gaia Data Release 1. Summary of the astrometric, photometric, and survey properties. *Astronomy and Astrophysics*, 595:A2, November 2016. arXiv:1609.04172, doi:10.1051/0004-6361/201629512.
- [GPD⁺16] Gaia Collaboration, T. Prusti, J. H. J. de Bruijne, A. G. A. Brown, A. Vallenari, C. Babusiaux, C. A. L. Bailer-Jones, U. Bastian, M. Biermann, D. W. Evans, and et al. The Gaia mission. *Astronomy and Astrophysics*, 595:A1, November 2016. arXiv:1609.04153, doi:10.1051/0004-6361/201629272.
- [GZT⁺18] N. J. Goldbaum, J. A. ZuHone, M. J. Turk, K. Kowalik, and A. L. Rosen. unyt: Handle, manipulate, and convert data with units in Python. *ArXiv e-prints*, June 2018. arXiv:1806.02417.
- [LKD⁺09] Nicholas M. Law, Shrinivas R. Kulkarni, Richard G. Dekany, Eran O. Ofek, Robert M. Quimby, Peter E. Nugent, Jason Surace, Carl C. Grillmair, Joshua S. Bloom, Mansi M. Kasliwal, Lars

- Bildsten, Tim Brown, S. Bradley Cenko, David Ciardi, Ernest Croner, S. George Djorgovski, Julian van Eyken, Alexei V. Filippenko, Derek B. Fox, Avishay Gal-Yam, David Hale, Nouhad Hamam, George Helou, John Henning, D. Andrew Howell, Janet Jacobsen, Russ Laher, Sean Mattingly, Dan McKenna, Andrew Pickles, Dovi Poznanski, Gustavo Rahmer, Arne Rau, Wayne Rosing, Michael Shara, Roger Smith, Dan Starr, Mark Sullivan, Viswa Velur, Richard Walters, and Jeff Zolkower. The Palomar Transient Factory: System Overview, Performance, and First Results. *Publications of the Astronomical Society of the Pacific*, 121:1395, December 2009. [arXiv:0906.5350](https://arxiv.org/abs/0906.5350), [doi:10.1086/648598](https://doi.org/10.1086/648598).
- [LMG⁺18] Russ R. Laher, Frank J. Masci, Steve Groom, Benjamin Rusholme, David L. Shupe, Ed Jackson, Jason Surace, Dave Flynn, Walter Landry, Scott Terek, George Helou, Ron Beck, Eugene Hacquard, Umaa Rebbapragada, Brian Bue, Roger M. Smith, Richard G. Dekany, Adam A. Miller, S. B. Cenko, Eric Bellm, Maria Patterson, Thomas Kupfer, Lin Yan, Tom Barlow, Matthew Graham, Mansi M. Kasliwal, Thomas A. Prince, and Shrinivas R. Kulkarni. Processing Images from the Zwicky Transient Facility. *RTSE Conference Proceedings*, 2018. [arXiv:1708.01584](https://arxiv.org/abs/1708.01584).
- [PAA⁺16] Planck Collaboration, P. A. R. Ade, N. Aghanim, M. Arnaud, M. Ashdown, J. Aumont, C. Baccigalupi, A. J. Banday, R. B. Barreiro, J. G. Bartlett, N. Bartolo, E. Battaner, R. Battye, K. Benabed, A. Benoît, A. Benoit-Lévy, J. P. Bernard, M. Bersanelli, P. Bielewicz, J. J. Bock, A. Bonaldi, L. Bonavera, J. R. Bond, J. Borrill, F. R. Bouchet, F. Boulanger, M. Bucher, C. Burigana, R. C. Butler, E. Calabrese, J. F. Cardoso, A. Catalano, A. Challinor, A. Chamballu, R. R. Chary, H. C. Chiang, J. Chluba, P. R. Christensen, S. Church, D. L. Clements, S. Colombi, L. P. L. Colombo, C. Combet, A. Coullais, B. P. Crill, A. Curto, F. Cuttaia, L. Danese, R. D. Davies, R. J. Davis, P. de Bernardis, A. de Rosa, G. de Zotti, J. Delabrouille, F. X. Désert, E. Di Valentino, C. Dickinson, J. M. Diego, K. Dolag, H. Dole, S. Donzelli, O. Doré, M. Douspis, A. Ducout, J. Dunkley, X. Dupac, G. Efstathiou, F. Elsner, T. A. Enßlin, H. K. Eriksen, M. Farhang, J. Ferguson, F. Finelli, O. Forni, M. Frailis, A. A. Fraisse, E. Franceschi, A. Frejsel, S. Galeotta, S. Galli, K. Ganga, C. Gauthier, M. Gerbino, T. Ghosh, M. Giard, Y. Giraud-Héraud, E. Giusarma, E. Gjerløw, J. González-Nuevo, K. M. Górski, S. Gratton, A. Gregorio, A. Gruppuso, J. E. Gudmundsson, J. Hamann, F. K. Hansen, D. Hanson, D. L. Harrison, G. Helou, S. Henrot-Versillé, C. Hernández-Monteagudo, D. Herranz, S. R. Hildebrandt, E. Hivon, M. Hobson, W. A. Holmes, A. Hornstrup, W. Hovest, Z. Huang, K. M. Huffenberger, G. Hurier, A. H. Jaffe, T. R. Jaffe, W. C. Jones, M. Juvela, E. Keihänen, R. Keskitalo, T. S. Kisner, R. Kneissl, J. Knoche, L. Knox, M. Kunz, H. Kurki-Suonio, G. Lagache, A. Lähteenmäki, J. M. Lamarre, A. Lasenby, M. Lattanzi, C. R. Lawrence, J. P. Leahy, R. Leonardi, J. Lesgourgues, F. Levrier, A. Lewis, M. Liguori, P. B. Lilje, M. Linden-Vørnle, M. López-Cañiego, P. M. Lubin, J. F. Macías-Pérez, G. Maggio, D. Maino, N. Mandolesi, A. Mangilli, A. Marchini, M. Maris, P. G. Martin, M. Martinelli, E. Martínez-González, S. Masi, S. Matarrese, P. McGehee, P. R. Meinhold, A. Melchiorri, J. B. Melin, L. Mendes, A. Mennella, M. Migliaccio, M. Millea, S. Mitra, M. A. Miville-Deschênes, A. Moneti, L. Montier, G. Morgante, D. Mortlock, A. Moss, D. Munshi, J. A. Murphy, P. Naselsky, F. Nati, P. Natoli, C. B. Netterfield, H. U. Nørgaard-Nielsen, F. Novello, D. Novikov, I. Novikov, C. A. Oxborrow, F. Paci, L. Pagano, F. Pajot, R. Paladini, D. Paoletti, B. Partridge, F. Pasian, G. Patanchon, T. J. Pearson, O. Perdereau, L. Perotto, F. Perrotta, V. Pettorino, F. Piacentini, M. Piat, E. Pierpaoli, D. Pietrobon, S. Plaszczynski, E. Pointecouteau, G. Polenta, L. Popa, G. W. Pratt, G. Prézeau, S. Prunet, J. L. Puget, J. P. Rachen, W. T. Reach, R. Rebolo, M. Reinecke, M. Remazeilles, C. Renault, A. Renzi, I. Ristorcelli, G. Rocha, C. Rosset, M. Rossetti, G. Roudier, B. Rouillé d'Orfeuil, M. Rowan-Robinson, J. A. Rubiño-Martín, B. Rusholme, N. Said, V. Salvatelli, L. Salvati, M. Sandri, D. Santos, M. Savelainen, G. Savini, D. Scott, M. D. Seiffert, P. Serra, E. P. S. Shellard, L. D. Spencer, M. Spinelli, V. Stolyarov, R. Stompor, R. Sudiwala, R. Sunyaev, D. Sutton, A. S. Suur-Uski, J. F. Sygnet, J. A. Tauber, L. Terenzi, L. Toffolatti, M. Tomasi, M. Tristram, T. Trombetti, M. Tucci, J. Tuovinen, M. Türler, G. Umata, L. Valenziano, J. Valiviita, F. Van Tent, P. Vielva, F. Villa, L. A. Wade, B. D. Wandelt, I. K. Wehus, M. White, S. D. M. White, A. Wilkinson, D. Yvon, A. Zacchei, and A. Zonca. Planck 2015 results. XIII. Cosmological parameters. *Astronomy and Astrophysics*, 594:A13, September 2016. [doi:10.1051/0004-6361/201525830](https://doi.org/10.1051/0004-6361/201525830).
- [STS13] STScI Development Team. ppsynphot: Synthetic photometry software package. *Astrophysics Source Code Library*, March 2013. [arXiv:1303.023](https://arxiv.org/abs/1303.023).
- [TPS⁺18] The Astropy Collaboration, A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, A. Ginsburg, J. T. VanderPlas, L. D. Bradley, D. Pérez-Suárez, M. de Val-Borro, T. L. Aldcroft, K. L. Cruz, T. P. Robitaille, E. J. Tollerud, C. Ardelean, T. Babej, M. Bachetti, A. V. Bakanov, S. P. Bamford, G. Barentsen, P. Barmby, A. Baumbach, K. L. Berry, F. Biscani, M. Boquien, K. A. Bostroem, L. G. Bouma, G. B. Brammer, E. M. Bray, H. Breytenbach, H. Buddelmeijer, D. J. Burke, G. Calderone, J. L. Cano Rodríguez, M. Cara, J. V. M. Cardoso, S. Cheedella, Y. Copin, D. Crichton, D. DÁvella, C. Deil, É. Depagne, J. P. Dietrich, A. Donath, M. Droettboom, N. Earl, T. Erben, S. Fabbro, L. A. Ferreira, T. Finethy, R. T. Fox, L. H. Garrison, S. L. J. Gibbons, D. A. Goldstein, R. Gommers, J. P. Greco, P. Greenfield, A. M. Groener, F. Grollier, A. Hagen, P. Hirst, D. Homeier, A. J. Horton, G. Hosseinzadeh, L. Hu, J. S. Hunkeler, Ž. Ivezić, A. Jain, T. Jenness, G. Kanarek, S. Kendrew, N. S. Kern, W. E. Kerzendorf, A. Khvalko, J. King, D. Kirkby, A. M. Kulkarni, A. Kumar, A. Lee, D. Lenz, S. P. Littlefair, Z. Ma, D. M. Macleod, M. Mastroiello, C. McCully, S. Montagnac, B. M. Morris, M. Mueller, S. J. Mumford, D. Muna, N. A. Murphy, S. Nelson, G. H. Nguyen, J. P. Ninan, M. Nöthe, S. Ogaz, S. Oh, J. K. Parejko, N. Parley, S. Pascual, R. Patil, A. A. Patil, A. L. Plunkett, J. X. Prochaska, T. Rastogi, V. Reddy Janga, J. Sabater, P. Sakurikar, M. Seifert, L. E. Sherbert, H. Sherwood-Taylor, A. Y. Shih, J. Sick, M. T. Silbiger, S. Singanamalla, L. P. Singer, P. H. Sladen, K. A. Sooley, S. Sornarajah, O. Streicher, P. Teuben, S. W. Thomas, G. R. Tremblay, J. E. H. Turner, V. Terrón, M. H. van Kerkwijk, A. de la Vega, L. L. Watkins, B. A. Weaver, J. B. Whitmore, J. Woillez, and V. Zabalza. The Astropy Project: Building an inclusive, open-science project and status of the v2.0 core package. *ArXiv e-prints*, page arXiv:1801.02634, January 2018. [arXiv:1801.02634](https://arxiv.org/abs/1801.02634).

EarthSim: Flexible Environmental Simulation Workflows Entirely Within Jupyter Notebooks

Dharhas Pothina^{‡*}, Philipp J. F. Rudiger[§], James A Bednar[§], Scott Christensen^{‡†}, Kevin Winters^{‡†}, Kimberly Pevey^{‡†}, Christopher E. Ball^{§†}, Gregory Brener^{§†}

https://youtu.be/KTbd_oUkP4Q



Abstract—Building environmental simulation workflows is typically a slow process involving multiple proprietary desktop tools that do not interoperate well. In this work, we demonstrate building flexible, lightweight workflows entirely in Jupyter notebooks. We demonstrate these capabilities through examples in hydrology and hydrodynamics using the AdH (Adaptive Hydraulics) and GSSHA (Gridded Surface Subsurface Hydrologic Analysis) simulators. The goal of this work is to provide a set of tools that work well together and with the existing scientific python ecosystem, that can be used in browser based environments and that can easily be reconfigured and repurposed as needed to rapidly solve specific emerging issues such as hurricanes or dam failures.

As part of this work, extensive improvements were made to several general-purpose open source packages, including support for annotating and editing plots and maps in Bokeh and HoloViews, rendering large triangular meshes and regridding large raster data in HoloViews, GeoViews, and Dashader, and widget libraries for Param. In addition, two new open source projects are being released, one for triangular mesh generation (Filigree) and one for environmental data access (Quest).

Index Terms—python, visualization, workflows, environmental simulation, hydrology, hydrodynamics, grid generation

Introduction

Environmental Simulation consists of using historical, current and forecasted environmental data in conjunction with physics-based numerical models to simulate conditions at locations across the globe. The simulations of primary interest are weather, hydrology, hydrodynamics, soil moisture and groundwater transport. These simulations combine various material properties such as soil porosity and vegetation types with topology such as land surface elevation and bathymetry, along with forcing functions such as rainfall, tide, and wind, to predict quantities of interest such as water depth, soil moisture, and various fluxes. Currently, the primary methodology to conduct these simulations requires a combination of heavy proprietary desktop tools such as Surface-water Modeling System (SMS) [Aquaveo] and Computational Model Builder (CMB) [Hines09], [CMB] that are tied to certain platforms and do not interoperate well with each other.

* Corresponding author: Dharhas.Pothina@erdc.dren.mil

‡ US Army Engineer Research and Development Center

§ Anaconda, Inc.

† These authors contributed equally.

Copyright © 2018 Dharhas Pothina et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

The process of building and running environmental simulations using these tools is time consuming, requiring a large amount of manual effort and a fair amount of expertise. Typically, the time required to build a reasonable model is measured in months. These workflows support some use cases well, especially multi-year projects where there is often the need for highly accurate, high-resolution physical modeling. But these existing tools and workflows are too heavyweight for other potential applications, such as making short-term operational decisions in novel locations. They also make it difficult to flexibly switch between desktop and remote high-performance-computing (HPC) systems as needed for scaling up and for interactive use.

An additional limitation of the existing desktop tools (i.e. CMB and SMS) are that the users are limited to the functionality and algorithms that are available in the tool. Adding new functionality requires expensive development efforts as well as cooperation of the tool vendors. For example, adding a coastline extraction tool to CMB based on the grabcut algorithm [Carsten04] required contracting with the vendor and several months of development time. As shown later in this paper, the functionality can be quickly put together using existing packages within the scientific python ecosystem.

In this work, we demonstrate building flexible, lightweight workflows entirely in Jupyter notebooks with the aim of timely support for operational decisions, providing basic predictions of environmental conditions quickly and flexibly for any region of the globe. For small datasets these notebooks can operate entirely locally, or they can be run with local display and remote computation and storage for larger datasets. We demonstrate these capabilities through examples in hydrology and hydrodynamics using the AdH [McAlpin17] and GSSHA [Downer08] simulators. The goal of this work is to provide a set of tools that work well together and with the existing scientific python ecosystem, can be used in browser based environments and that can easily be reconfigured and repurposed as needed to rapidly solve specific emerging issues. A recent example of this was during Hurricane Harvey when ERDC was required at short notice to provide flood inundation simulations of the cities of San Antonio, Houston and Corpus Christi to emergency response personnel. This required rapid assembly of available data from disparate sources, generation of computational grids, model setup and execution as well as generation of custom output visualizations.

An explicit decision was made to avoid creation of new

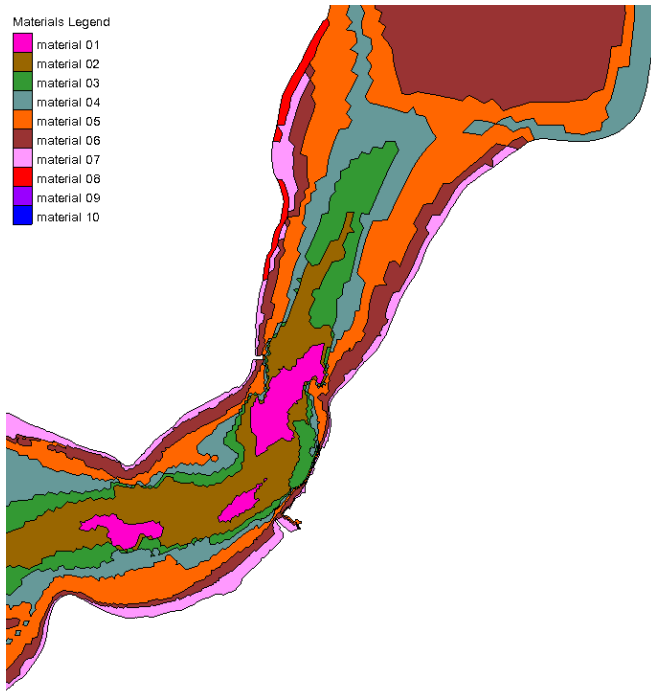


Fig. 1: Example of a Region of Interest sectioned into multiple polygons each with a specific material property.

special-purpose libraries as much as possible and to instead enhance existing tools with the capabilities required. Hence, as part of this work, extensive improvements were made to several general-purpose open source packages, including support for annotating and editing plots and maps in Bokeh and HoloViews, rendering large triangular meshes and regridding large raster data in HoloViews, GeoViews, and Datashader, and widget libraries for Param [Bokeh], [Holoviews], [Geoviews], [Datashader], [Param]. In addition, two new open source projects are being released for triangular mesh generation and environmental data access [Filigree], [Quest].

Background

The traditional workflow for building environmental simulations can be broken down into the following stages:

- 1) **Model specification:** Building a human-specified conceptual model that denotes regions of interest (ROIs) and their properties. Typically, this involves drawing of points, lines and polygons to define the ROIs and define features, boundary types and material properties (land surface elevation, soil type, bottom friction, permeability, etc.). See Figure 1.
- 2) **Data Retrieval:** Material properties, hydrology and climatology datasets are retrieved from various public web-based and local-data stores.
- 3) **Computational mesh generation:** The ROIs are partitioned into a computational mesh that is used by the environmental simulation engine. The simulation types that we are focused on in this work use a 2D structured/regular rectangular grid or an unstructured 2D triangular mesh (See Figure 2). 3D meshes are obtained by extruding the 2D mesh in the z direction in the form of layers.

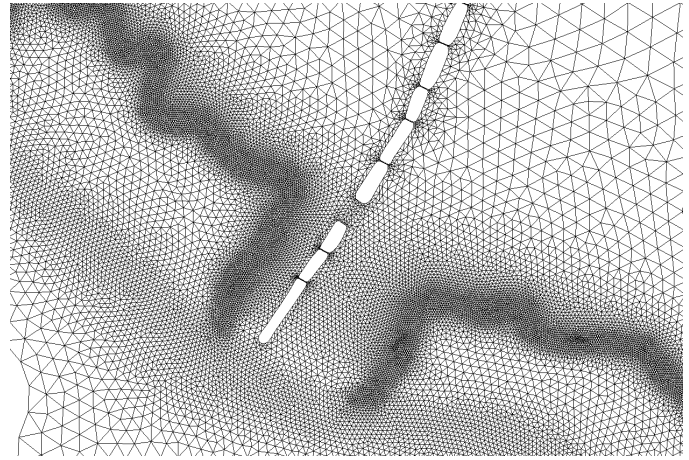


Fig. 2: Example of an unstructured 2D triangular computational mesh of a river that is transected by a roadway embankment with culvert and bridge openings.

Initial generation of a computational mesh is typically automated and controlled by attributes in the model specification process. After this an iterative approach is used to build a high-quality mesh based on the needs of the numerical algorithms and to resolve key physical properties in certain regions. Often mesh vertices and elements need to be adjusted manually.

- 4) **Data gridding:** Based on the model specification, any spatially varying material properties, initial conditions and time-varying forcing functions (i.e. boundary conditions) are regridded from the original data sources to the computational mesh.
- 5) **Simulation:** The computational mesh along with the re-gridded data, plus any model parameters (turbulence model, etc.) and forcings required (rainfall, etc.) needed for a specific simulation are written to files formatted for a particular environmental simulation engine. This model is then run with the simulation engine (i.e. AdH, GSSHA). For larger simulations, this is run on an HPC system.
- 6) **Visualization/analysis:** The results of environmental simulations typically consist of time varying scalar and vector fields defined on the computational mesh, stored in binary or ASCII files. Analysts first render an overall animation of each quantity as a sanity check, typically in 2D or 3D via a VTK-based Windows app in current workflows. For more detailed analysis, analysts typically specify certain lower-dimensional subsets of this multidimensional space, such as:
 - **Virtual measurement stations:** A specific point on the Earth's surface where e.g. water level can be computed for every time point and then compared with historical data from nearby actual measurement stations
 - **Cross-sections:** A 1D curve across the surface of the Earth, where a vertical slice can be extracted and plotted in 2D
 - **Iso-surfaces:** Slices through the multidimensional data where a certain value is held constant, such as salinity. Associated quantities (e.g. temperature) can then be plotted in 2D as a color.

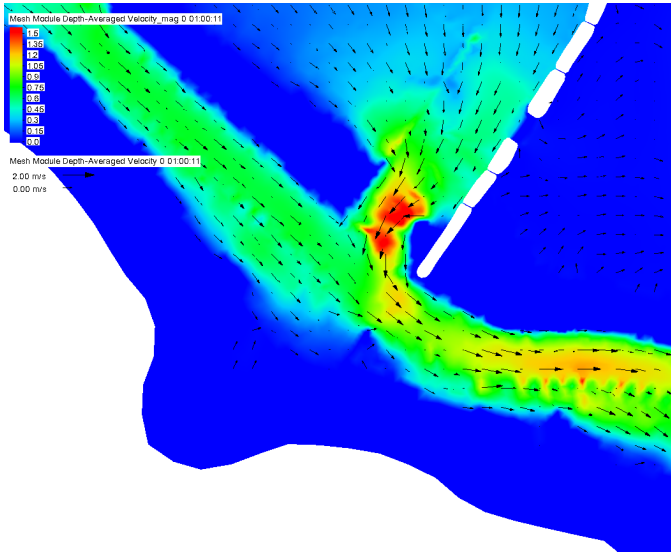


Fig. 3: Water velocity color contours overlain with velocity quiver plot showing river flow bypassing roadway embankment.

Figure 3 shows an example visualization of a water circulation field.

This overall pipeline can give very high quality results, but it takes 3-6 months to build and run a model, which is both expensive and also precludes the use of this approach for modeling emergent issues quickly enough to affect operational decisions. Most of these stages are also locked into particular Windows-based GUI applications that are typically tied to execution only on specific desktop machines where they are installed. In most cases, once the model input files are generated, they can be manually moved to an HPC cluster and run from the command line, but then no GUI is available. This linkage of computation and visualization can be very problematic, because the local machine may not have enough processing power to simulate the model in a reasonable time, but if the model is simulated remotely, the resulting data files can be too large to be practical to transfer to the local machine for analysis. To give an example of the data sizes and timescales involved, simple example/tutorial hydrodynamic model runs on idealized domains using AdH can take up to an hour. The largest simulation that can be run on a local workstation generate files of the order of a few gigabytes and can take several days to run. Realistic, regional scale models are almost always run on HPC systems typically using 500 to a 1000 processors and generate up to a terabyte worth of data. HPC runs typically take anywhere from several hours to a day to complete. An example of the type of HPC systems used for AdH model runs are the Department of Defences supercomputers Topaz and Onyx. Topaz is an SGI ICE X System. Standard compute nodes have two 2.3-GHz Intel Xeon Haswell 18-core processors (36 cores) and 128 GBytes of DDR4 memory. Compute nodes are interconnected by a 4x FDR InfiniBand Hypercube network. Onyx is a Cray XC40/50. Standard compute nodes have two 2.8-GHz Intel Xeon Broadwell 22-core processors (44 cores) and 128 GBytes of DDR4 memory. Compute nodes are interconnected by a Cray Aries high-speed network. Both systems have dedicated GPU compute nodes available. [ERDCHPC]

Moreover, the tools that implement the current workflow are primarily "heavyweight" approaches that encode a wide set of assumptions and architectural decisions specific to the applica-

tion domain (environmental simulation), and changing any of these assumptions or decisions will typically require an extensive vendor-implemented project of C/C++ software development. These constraints make it difficult for end users who are experts in the application domain (but not necessarily full-time software developers) to develop and test architectural improvements and the effects of different modeling approaches that could be suitable for specific applications.

Because much of the functionality required to implement the above workflow is already available as general-purpose libraries in the Python software ecosystem, we realized that it was feasible to provide a lightweight, flexible alternative for most of these stages, with rapid iterative refinement of a conceptual model, simulation on whatever hardware is available, and fast, flexible, primarily 2D visualization of remote or local data in a local browser. The idea is to put power and flexibility into the hands of domain experts so that they can respond quickly and easily to emerging issues that require input to help decision making throughout their organizations, without requiring a lengthy period of model development and without requiring external software contractors to make basic changes to assumptions and modeling mechanisms. In this paper, we show how we have built such a system.

EarthSim

EarthSim is a website and associated GitHub repository that serves two purposes. First, it is a location to work on new tools before moving them into other more general purpose python libraries as they mature. Second, it contains examples of how to solve the common Earth Science simulation workflow and visualization problems outlined above. EarthSim aims to demonstrate building flexible, lightweight workflows entirely in Jupyter notebooks with the goal of timely support for operational decisions, providing basic predictions of environmental conditions quickly and flexibly for any region of the globe. The overall goal is to provide a set of tools that work well together and with the wider scientific python ecosystem. EarthSim is not meant to be a one-size-fits-all solution for environmental simulation workflows but a library of tools that can be mixed and matched with other tools within the python ecosystem to solve problems flexibly and quickly. To that end, the specific enhancements we describe are targeted towards areas where existing tools were not available or were insufficient for setting up an end to end simulation.

EarthSim primarily consists of the core PyViz tools (Bokeh, HoloViews, GeoViews, Datashader, and Param) as well as two other new open source tools Filigree and Quest. Short descriptions of these tools follow:

Bokeh provides interactive plotting in modern web browsers, running JavaScript but controlled by Python. Bokeh allows Python users to construct interactive plots, dashboards, and data applications without having to use web technologies directly.

HoloViews provides declarative objects for instantly visualizable data, building Bokeh plots from convenient high-level specifications so that users can focus on the data being explored.

Datashader allows arbitrarily large datasets to be rendered into a fixed-size raster for display, making it feasible to work with large and remote datasets in a web browser, either in batch mode using Datashader alone or interactively when combined with HoloViews and Bokeh.

Param allows the declaration of user-modifiable values called Parameters that are Python attributes extended to have features

such as type and range checking, dynamically generated values, documentation strings, and default values. Param allows code to be concise yet robustly validated, while supporting automatic generation of widgets for configuration setting and for controlling visualizations (e.g. using ParamBokeh).

All of the above tools are fully general, applicable to *any* data-analysis or visualization project, and establish a baseline capability for running analysis and visualization of arbitrarily large datasets locally or remotely, with fully interactive visualization in the browser regardless of dataset size (which is not true of most browser-based approaches). The key is concept is that the local client system will always be capable of performing the visualization, i.e. can deliver it to the user in a browser, regardless of the dataset size. The assumption is that the remote server will be able to handle the datasets, but because Datashader is based on the Dask parallel library, it is possible to assemble a remote system out of as many nodes as required need to handle a given dataset, also work can be done out of core if the user is prepared to wait. Based on this architecture, this software stack will not be a limiting factor, only the users' ability to procure nodes or the time taken to render. This is in contrast to other software stacks that typically have a hard size limit. It can be clarified that we have achieved this claim by a three-level implementation: Dask, which can distribute the computation across arbitrarily many user-selected nodes (or multiplexed over time using the same node) to achieve the required computational power and memory, Datashader, which can make use of data and compute managed by dask to reduce the data into a fixed-size raster for display, and Bokeh, to render the resulting raster along with other relevant data like maps.

In addition, the data is not encoded, compressed, modeled, or subsampled, it's just aggregated (no data is thrown away, it's simply summed or averaged), and the aggregation is done on the fly to fit the resolution of the screen. This provides the experience of having the dataset locally, without actually having it and allows for responsive interactive exploration of very large datasets.

The other libraries involved are specialized for geographic applications:

GeoViews extends HoloViews to support geographic projections using the Cartopy library, making it easy to explore and visualize geographical, meteorological, and oceanographic datasets.

Quest is a library that provides a standard API to search, publish and download data (both geographical and non-geographical) across multiple data sources including both local repositories and web based services. The library also allows provides tools to manipulate and manage the data that the user is working with.

Filigree is a library version of the computational mesh generator from Aquaveo's XMS software suite [Aquaveo]. It allows for the generation of high quality irregular triangular meshes that conform to the constraints set up by the user.

In surveying the landscape of existing python tools to conduct environmental simulations entirely within a Jupyter notebook environment, four areas were found to be deficient:

- 1) Interactively drawing and editing of glyphs (Points, Lines, Polygons etc) over an image or map.
- 2) Interactive annotation of objects on an image or map.
- 3) Efficient visualization of large structured and unstructured grid data in the browser.
- 4) Setup of interactive dashboards.

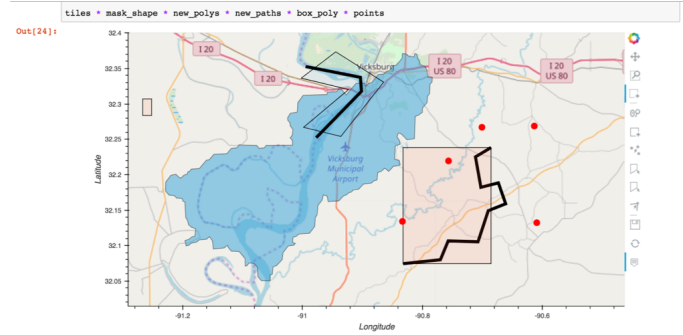


Fig. 4: Visualization of drawing tools showing drawn polygons, points, paths, and boundary boxes overlaying a web tile service.

```
In [9]: path_stream.element.data
Out[9]: OrderedDict({'Longitude',
                    array([-10129501, -10119319, -10119114, -10127543]),
                    ('Latitude', array([3809682, 3807677, 3805097, 3796477]))})
```

Fig. 5: Drawing tools provide a dynamic link to source data accessible via python backend.

In the next few sections, we describe how this functionality is now available from Python without requiring custom Javascript code.

Enhancements: Drawing Tools

The Bokeh plotting library has long supported extensive interactive operations for exploring existing data. However, it did not previously offer any facilities for generating or editing new data interactively, which is required when constructing inputs for running new simulations. In this project, we added a set of Bokeh editing/drawing tools (See Figure 4), which are sophisticated multi-gesture tools that can add, delete, or modify glyphs on a plot. The edit tools provide functionality for drawing and editing glyphs client-side (in the user's local browser) and synchronizing the changes with data sources on the Python server that can then be accessed in Python. The individual tools can be enabled as needed for each particular plot:

- **BoxEditTool:** Drawing, dragging and deleting rectangular glyphs.
- **PointDrawTool:** Adding, dragging and deleting point-like glyphs.
- **PolyDrawTool:** Drawing, selecting and deleting Polygon (patch) and Path (polyline) glyphs.
- **PolyEditTool:** Editing the vertices of one or more Polygon or Path glyphs.

To make working with these tools easy, HoloViews was extended to define "streams" that provide an easy bidirectional connection between the JavaScript plots and Python (See Figure 5). This allows for definition of geometries in Python and editing in the interactive plot, or creation/modification of geometries in the interactive plot with subsequent access of the data from Python for further processing.

Similar tools allow editing points, polygons, and polylines.

As a simple motivating example, drawing a bounding box on a map now becomes a simple 7-line program:

```
import geoviews as gv
import geoviews.tile_sources as gts
```

```
import holoviews.streams as hvs

gv.extension('bokeh')
box = gv.Polygons(hv.Box(0, 0, 1000000))
roi = hvs.BoxEdit(source=box)
gts.StamenTerrain.options(width=900, height=500) * box
```

In a Jupyter notebook, this code will display a world map and let the user move or edit a box to cover the region of interest (ROI), which can then be accessed from Python as:

```
roi.data
```

For example, Figure 6 demonstrates how USGS National Elevation Dataset (NED) data can then be retrieved for the ROI as:

```
import quest
import xarray as xr
import holoviews as hv
import cartopy.crs as ccrs

element = gv.operation.project(hv.Polygons(
    roi.element), projection=ccrs.PlateCarree()
)
xs, ys = element.array().T
bbox = list(gv.util.project_extents(
    (xs[0], ys[0], xs[2], ys[1]),
    ccrs.GOOGLE_MERCATOR,
    ccrs.PlateCarree())
)

collection_name = 'elevation_data'
quest.api.new_collection(name=collection_name)
service_features = quest.api.get_features(
    uris='svc://usgs-ned:19-arc-second',
    filters={'bbox': bbox}
)
collection_features = quest.api.add_features(
    collection=collection_name,
    features=service_features
)
datasets = quest.api.stage_for_download(
    uris=collection_features
)
quest.api.download_datasets(datasets=datasets)
elevation_dataset = quest.api.apply_filter(
    name='raster-merge',
    options={'datasets': datasets, 'bbox': bbox}
)['datasets'][0]
elevation_file = quest.api.get_metadata(
    elevation_dataset
)[elevation_dataset]['file_path']

elevation_raster = xr.open_rasterio(
    elevation_file
).isel(band=0)
img = gv.Image(elevation_raster, ['x', 'y'])
gts.StamenTerrain.options(width=600) * img
```

Enhancements: Annotations

The drawing tools allow glyphs to be created graphically, which is an essential first step in designing a simulation. The next step is then typically to associate specific values with each such glyph, so that the user can declare boundary conditions, parameter values, or other associated labels or quantities to control the simulation. Examples of how to do this are provided in EarthSim as "annotators", which show an editable table alongside the plot that has drawing tools (See Figure 7), allowing users to input text or numerical values to associate with each glyph. The table and plots are interlinked, such that editing either one will update the other, making it simple to edit data however is most convenient.

Using an annotator currently requires defining a new class to control the behavior, but work on simplifying this process is

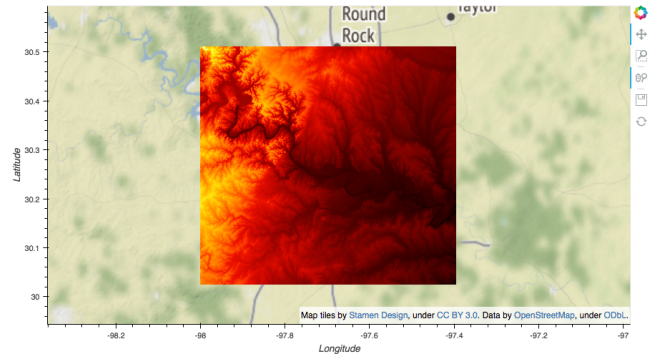


Fig. 6: Visualization data downloaded with quest for a ROI specified with the drawing tools.

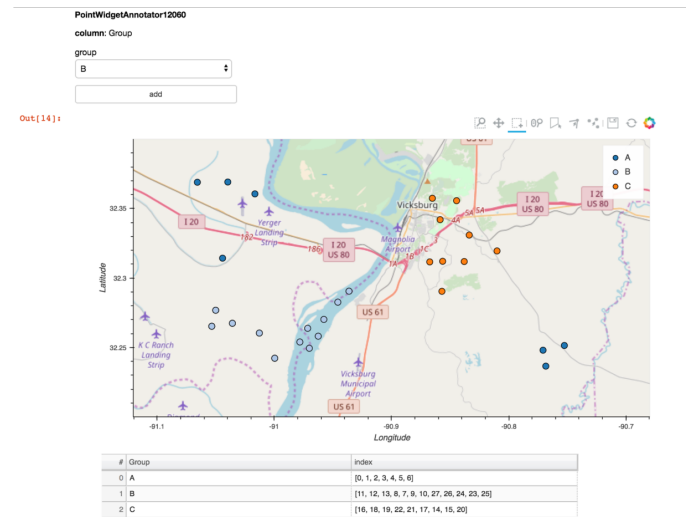


Fig. 7: The Point Annotation tool provides for indexing and grouping of points

ongoing, and if it can be made more straightforward the code involved will move into GeoViews or HoloViews as appropriate.

Enhancements: Efficient Raster regridding

Many of the datasets used in Earth-related workflows come in the form of multidimensional arrays holding values sampled regularly over some portion of the Earth's surface. These rasters are often very large and thus slow to transfer to a client browser, and are often too large for the browser to display at all. To make it feasible to work naturally with this data, efficient regridding routines were added to Datashader. Datashader is used by HoloViews to re-render data at the screen's resolution before display, requiring only this downsampled version to be transferred to the client browser. The raster support is described at datashader.org, using all available computational cores to quickly render the portions of the dataset needed for display. The same code can also be used to re-render data into a new grid spacing for a fixed-sized rectangular simulator like GSSHA.

The Datashader code does not currently provide reprojection of the data into a different coordinate system when that is needed. A separate implementation using the xESMF library was also developed for GeoViews to address this need and to provide additional Earth-specific interpolation options. The geoviews.org

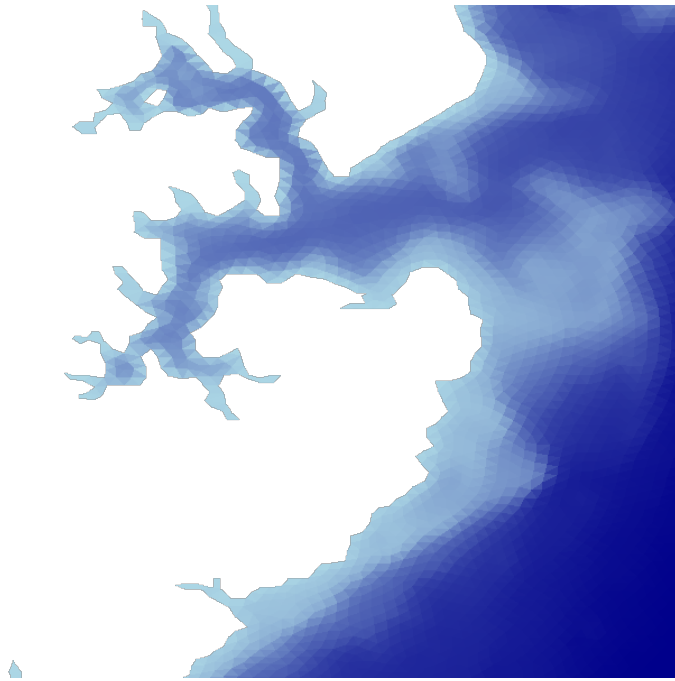


Fig. 8: Example of a datashader visualization of triangular unstructured mesh of a portion of Chesapeake Bay

[website](#) explains how to use either the Datashader or xESMF regridding implementations developed in this project.

Enhancements: Triangular mesh visualization

Although Earth imaging data is typically measured on a regular grid, how quickly the values change across the Earth’s surface is highly non-uniform. For instance, elevation changes slowly in many regions, but very quickly in others, and thus when simulating phenomena like water runoff it is often necessary to use very high resolution in some locations and relatively sparse sampling in others. To facilitate working with irregularly gridded data, the Bokeh, HoloViews, GeoViews, and Datashader libraries were extended to support "TriMesh" data, i.e., irregular triangle grids. For very large such grids, Datashader allows them to be rendered into much smaller rectangular grids for display, making it feasible to explore meshes with hundreds of millions of datapoints interactively. The other libraries provide additional interactivity for smaller meshes without requiring Datashader, while being able to use Datashader for the larger versions (Figure 8).

Interactive Dashboards

The drawing tools make it possible to generate interactive dashboards quickly and easily to visualize and interact with source data. Figure 9 shows hydrodynamic model simulation results displayed in an animation on the left. Users are able to query the results by annotating paths directly on the results visualization. As annotations are added, the drawing on the right dynamically updates to show the depth results along the annotated paths. The animation tool is dynamically linked to both drawings to demonstrate changes over time.

The drawing tools allow for specification of source data as key dimensions (independent variables or indices) or as value dimensions (dependent values or results data). Value dimensions can be visualized using widgets that are dynamically linked to

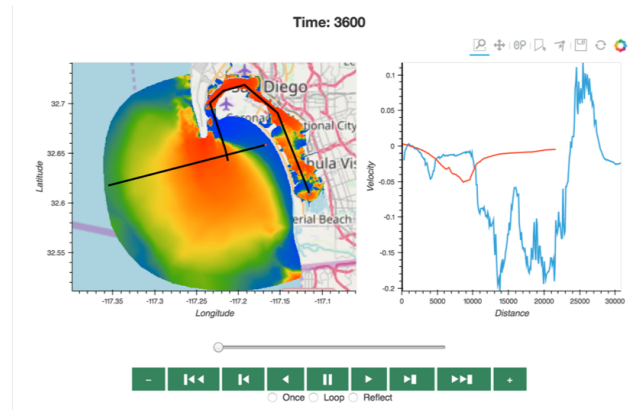


Fig. 9: Dashboard with animation demonstrating the ability to dynamically visualize multiple looks at a single source dataset.

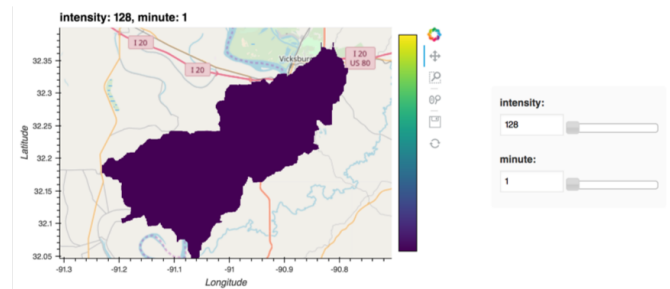


Fig. 10: Dynamic interaction with drawing via interactive widgets.

the drawing. This allows for simplified visualizations of multi-dimensional datasets such as parameter sweeps (Figure 10).

Drawings can be both the sender and receiver of dynamic information. Dashboards can be created that visualize data, allow users to specify paths in which to query data (e.g. river cross-sections), and visualize the results of the query in a dynamic manner. In Figure 11, the user-drawn cross-sections on the left query the underlying depth data and generate the image on the right. Users can then interact with the right image sliding the vertical black bar along the image which simultaneously updates the left image with a marker to denote the location along the path.

Crucially, note that very little of the code involved here is customized for hydrology or geographic applications specifically,

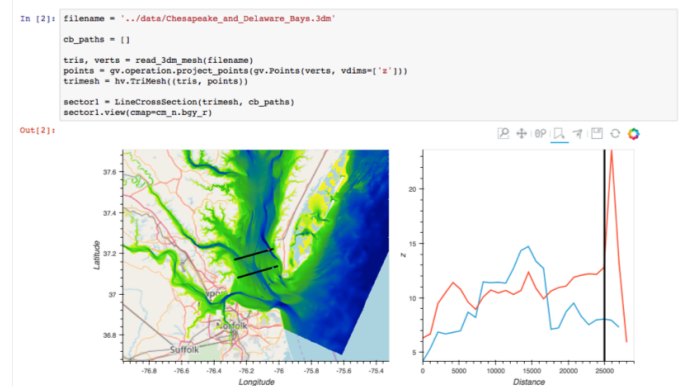


Fig. 11: Dynamic linking provides interaction between drawings as both sender and receiver.

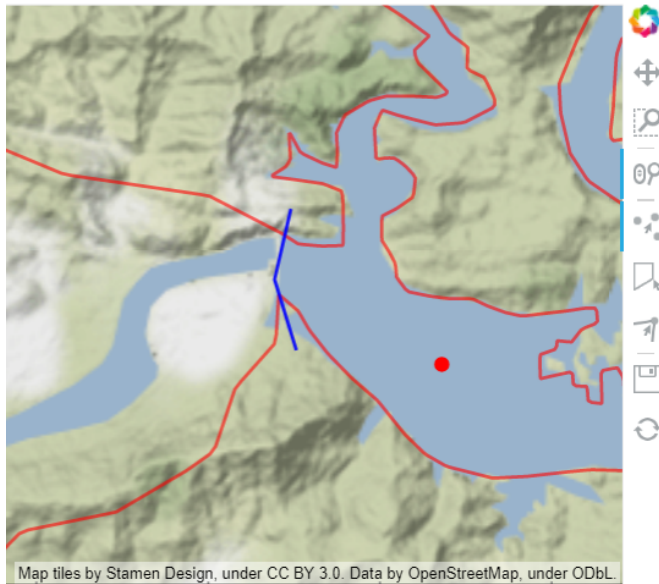


Fig. 12: User-specification of boundary, dam centerline, and reservoir level with the drawing tools.

which means that the same techniques can be applied to different problems as they arise in practice, even if they require changing the domain-specific assumptions involved.

GSSHA Hydrology Workflow Example

Using many of the tools described here, we have created a notebook workflow to setup, execute, and visualize the results of the GSSHA hydrology model. This workflow uses the drawing tools to specify an area of interest, and then Quest to download elevation and landuse data. Param is used to specify the model inputs, and finally GeoViews and Datashader are used to visualize the results. This flexible workflow can easily be applied to any location in the globe, and the specific output visualizations can easily be modified to meet specific project needs. The complete workflow can be found at http://earthsim.pyviz.org/topics/GSSHA_Workflow.html.

AdH Dambreak Workflow Example

The drawing tools, coupled with AdH, allow for rapid development of dambreak simulations to analyze potential hazard situations. In this example, as seen in Figure 12, the Polygon tool is used to delineate the boundary of a watershed, a dam centerline is specified with the Path tool, and a reservoir level specified with the Point tool.

Data from all three user-specified data sources can also be accessed and described via tables that are dynamically linked to the drawing. Additionally, Param widgets allow for users to specify the reservoir level as either a water depth or an elevation and whether to use an existing initial water depth file.

Available elevation data to describe the watershed is collected via Quest. Filigree is then called to develop a unstructured 2D triangular mesh within the boundary polygon. Using the basic information about the dam and the dynamically generated mesh, a reservoir is created behind the dam centerline. This is achieved by setting AdH water depths on the mesh to reflect the reservoir level. AdH then simulates the instantaneous breaching of the dam. The resulting simulation of water depths over time can then be visualized in the drawing tools as an animation.

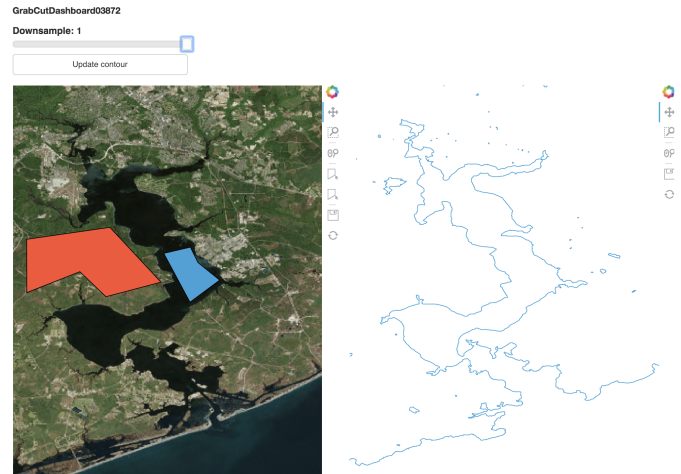


Fig. 13: Demonstration of a interactive widget for coastline extraction using the grabcut algorithm.

Coastline Extraction (GrabCut) Workflow Example

The GrabCut algorithm provides a way to annotate an image using polygons or lines to demark the foreground and background. The algorithm estimates the color distribution of the target object and that of the background using a Gaussian mixture model. This is used to construct a Markov random field over the pixel labels, with an energy function that prefers connected regions having the same label, and running a graph cut based optimization to infer their values. This procedure is repeated until convergence, resulting in an image mask denoting the foreground and background.

In this example this algorithm is applied to satellite imagery to automatically extract a coast- and shoreline contour. First we load an Image or RGB and wrap it in a HoloViews element, then we can declare a GrabCutDashboard (See Figure 13). Once we have created the object we can display the widgets using ParamBokeh, and call the view function to display some plots.

The toolbar in the plot on the left contains two polygon/polyline drawing tools to annotate the image with foreground and background regions respectively. To demonstrate this process in a static paper there are already two polygons declared, one marking the sea as the foreground and one marking the land as the background.

We can trigger an update in the extracted contour by pressing the Update contour button. To speed up the calculation we can also downsample the image before applying the Grabcut algorithm. Once we are done we can view the result in a separate cell. See Figure 14

The full coastline extraction with Grabcut Jupyter notebook is available at the EarthSim website: <https://pyviz.github.io/EarthSim/topics/GrabCut.html>

Future Work

Through the work presented here, we have shown that it is possible to build flexible, lightweight workflows entirely within Jupyter notebooks. However, there is still room for improvement. Current areas being targeted for development are:

- Performance enhancements for GIS & unstructured mesh datasets
- Making annotation and drawing tools easier to use (i.e. requiring less custom code)

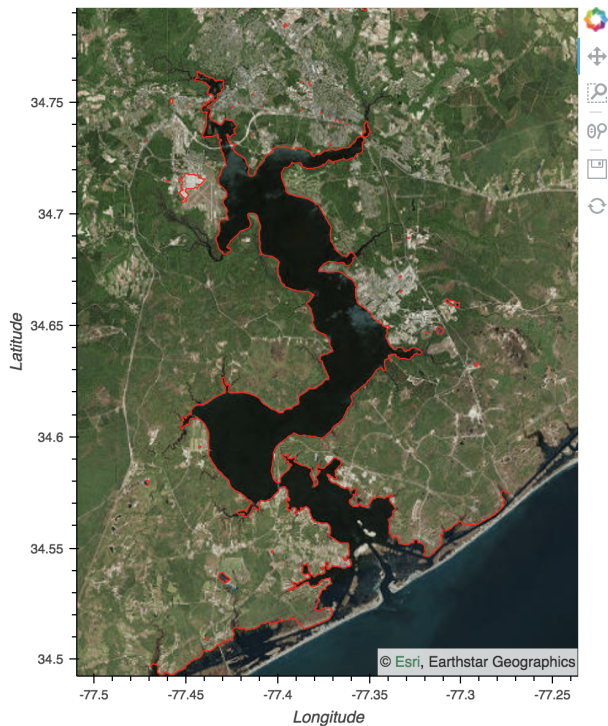


Fig. 14: Final image with extracted coastline show in red.

- Layout of Jupyter Notebooks in Dashboard type form factors with code hidden
- Integration with non Jupyter notebook web frontends (i.e. Tethys Platform [Swain14])
- Prototype bidirectional visual programming environment (e.g. ArcGIS Model Builder)

REFERENCES

- [Downer08] Downer, C. W., Ogden, F. L., and Byrd, A.R. 2008, GSSHAWIKI User's Manual, Gridded Surface Subsurface Hydrologic Analysis Version 4.0 for WMS 8.1, ERDC Technical Report, Engineer Research and Development Center, Vicksburg, Mississippi.
- [McAlpin17] McAlpin, J. T. 2017, Adaptive Hydraulics 2D Shallow Water (AdHSW2D) User Manual (Version 4.6), Engineer Research and Development Center, Vicksburg, Mississippi. Available at <https://chl.ercd.dren.mil/chladh>
- [Hines09] Hines, A. et al., Computational Model Builder (CMB): A Cross-Platform Suite of Tools for Model Creation and Setup, 2009 DoD High Performance Computing Modernization Program Users Group Conference, San Diego, CA, 2009, pp. 370-373.
- [Carsten04] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. 2004. "GrabCut": interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.* 23, 3 (August 2004), 309-314. DOI: <https://doi.org/10.1145/1015706.1015720>
- [Aquaveo] "Introduction | Aquaveo.com." [Online]. Available: <https://www.aquaveo.com/>. [Accessed: 05-Jul-2018].
- [CMB] "CMB Hydro | CMB." [Online]. Available: <https://www.computationalmodelbuilder.org/cmb-hydro/>. [Accessed: 05-Jul-2018].
- [Bokeh] "Welcome to Bokeh — Bokeh 0.13.0 documentation." [Online]. Available: <https://bokeh.pydata.org/en/latest/>. [Accessed: 05-Jul-2018].
- [HoloViews] "HoloViews — HoloViews." [Online]. Available: <http://holoviews.org/>. [Accessed: 05-Jul-2018].
- [Geoviews] "GeoViews — GeoViews 1.5.0+g63ddd7c-dirty documentation." [Online]. Available: <http://geoviews.org/>. [Accessed: 05-Jul-2018].
- [Datashader] "Installation — Datashader 0.6.6+geb9218c-dirty documentation." [Online]. Available: <http://datashader.org/>. [Accessed: 05-Jul-2018].
- [Param] "Param — Param 1.4.1-dev documentation." [Online]. Available: <http://param.pyviz.org/>. [Accessed: 05-Jul-2018].
- [Filigree] TODO talk to Aquaveo for correct Filigree reference
- [Quest] "Welcome to Quest's documentation! — Quest 0.5 documentation." [Online]. Available: <https://quest.readthedocs.io/en/latest/>. [Accessed: 05-Jul-2018].
- [EarthSim] "EarthSim — EarthSim 0.0.1 documentation." [Online]. Available: <http://earthsim.pyviz.org/>. [Accessed: 05-Jul-2018].
- [ERDCHPC] "ERDC DSRC - Hardware." [Online]. Available: <https://www.ercd.hpc.mil/hardware/index.html>. [Accessed: 05-Jul-2018].
- [Swain14] Swain, N., S. Christensen, N. Jones, and E. Nelson (2014), Tethys: A Platform for Water Resources Modeling and Decision Support Apps, paper presented at AGU Fall Meeting Abstracts.

Safe handling instructions for missing data

Dillon Niederhut^{‡*}

<https://youtu.be/2gkw2T5jAfo>

Abstract—In machine learning tasks, it is common to handle missing data by removing observations with missing values, or replacing missing data with the mean value for its feature. To show why this is problematic, we use listwise deletion and mean imputing to recover missing values from artificially created datasets, and we compare those models against ones with full information. Unless quite strong independence assumptions are met, we observe large biases in the resulting coefficients and an increase in the model's prediction error. We include a set of recommendations for handling missing data safely, and a case study showing how to put those recommendations into practice.

Index Terms—data science, missing data, imputation

Introduction

It is common in data analytics tasks to encounter missing values in datasets, where by *missing* we mean that some particular values does or should exist, and failed to be observed or recorded [little-rubin-2002]. There are several causes for missingness in datasets, which vary in theoretical difficulty from bit flipping (less problematic) to participant dropout in extended experimental studies (more problematic). According to the Inglis Conjecture¹, the best way to handle missing data is to apply corrections to the acquisition of that data. Because strategies for doing this tend to be domain specific, we will not be addressing this topic further in this paper.

In a similar vein, different research fields tend to have idiosyncratic methods for statistical correction of missing data, although they should not [newman-2014]. At one end of this spectrum is the epidemiology community, who are both unusual and commendable for their principled stance and clear guidelines regarding the handling and reporting of missingness [perkins-et-al-2018]. At the other end of the spectrum are research communities who handle missingness ad hoc, and frequently fail to report the presence of missingness at all.

Safe handling instructions are needed because the presence of unobserved data causes two theoretical problems with statistical models [schafer-graham-2002]. One of these is inferential: when data are missing from a particular feature or variable, estimates of the variance of that feature are unreliable and therefore so are any tests that use those variances. The second of these is descriptive: when data are missing according to a particular pattern, model

parameters that learn from the remaining data become biased by that pattern.

Anecdotally, the machine learning community appears less concerned with statistical inference, and feels relatively comfortable with the idea of replacing missing values with the mean value for each feature. The justification appears to be that mean imputing (called single imputation in the missingness literature) preserves the central tendency for that feature. However, statistical learning procedures are not determined by the mean values of their features—indeed, we often scale these down to zero—but rather by the the relationship between the variance of two features, which is modified by the presence of missing values and collapsed by single imputation.

This is the key theoretical problem with missing values: that they modify the covariance in datasets. To illustrate this, let's imagine that we have a dataset with no missing values and a linear relationship between one feature and one target. We'll remove 30% of the data (specifically the records at low values of our target) and then run a few models on the fully attested dataset and the dataset with missingness to see how the models compare.

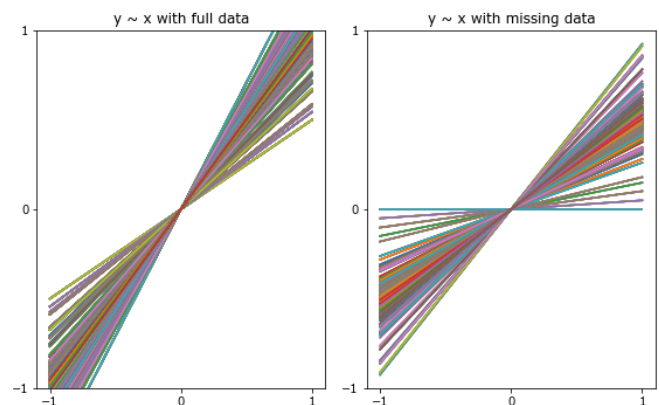


Fig. 1: Best fit lines across experiments for a fully-attested dataset compared to best fit lines and for a dataset with missingness imposed at random. Note the bias in the differences in slope.

What we see in Fig. 1 is that the models run over the dataset with missing values have a very particular kind of error. Specifically, the error in estimating the coefficient of the feature is not distributed as a Gaussian around the true value, but is always a reduction of the true value. This is bias, and it was created by the missingness process that we imposed on our data.

There are a theoretically infinite number of physical process that generate missingness in datasets, so in practice we will bin

* Corresponding author: dniederhut@enthought.com

‡ Enthought, Inc.

them into one of three categories that characterizes the input to the process which generates those missing values [rubin-1976]. If the probability that a value is missing is independent of any input, the process is stochastic and we call it Missing Completely At Random (MCAR). If the probability that a value is missing depends on another feature in our dataset, we call this Missing At Random (MAR)². If the probability that a value is missing depends on that value itself, we call this Missing Not At Random (MNAR). In theory, it is not possible to be certain whether values are MAR or MNAR, so they tend to be treated similarly.

Methods

To demonstrate when and how this bias appears, 1890 datasets were randomly generated with linear, quadratic, and sinusoidal relationships between two features and one target, at sizes that ranged between 100 and 10000 rows, and with an error term that varied in strength between factors of 0.0 (no error) and 0.5 (half of the magnitude of the data). Missingness regimes were imposed on only one of these two features, which we will refer to as the principle feature (x). The equations for generating the targets are given in Eqs. 1, 2, and 3. The datasets were supplemented with two auxiliary features³ whose correlation strength with the principal feature varied between 0.0 and 0.5.

$$t = 2 * x + y + \epsilon \quad (1)$$

$$t = x^2 - y + \epsilon \quad (2)$$

$$t = 2 * \sin(x) - y + \epsilon \quad (3)$$

A fractional amount of values was removed from the principal feature for each of the three missingness regimes, MCAR, MAR, and MNAR. For data missing completely at random, this was done with `np.random.choice`. For data missing at random and not at random, this was done by using the index of the N smallest values of the target and the principle feature, respectively. The amount of data removed varied between 0% (no missingness) and 50% of attested values, which is typical of the amount of missingness reported in experimental studies (50% is on the high end, more likely to be observed in longitudinal studies [sullivan-et-al-2017]).

Missing values were corrected using three different strategies. The first of these was to remove entire rows where any data is non present—this is called listwise deletion. The second was single imputation. We used the mean imputer from scikit-learn, but prior research shows that more complicated single imputation (like using the per sample grouped mean) has the same theoretical problems. The third strategy was an expectation maximization routine implemented in impute [impyute], which estimates replacements for missing values that maximize the probability of the rest of the data.

These datasets were fit with four models—linear regression, lasso regression, ridge regression, and support vector regression from scikit-learn. For stability when generating statistical summaries, each experimental combination for datasets with less than 10,000 rows was run through ten trials. This resulted in a total of 3,628,800 experiments.

For each experiment, difference scores were calculated for model coefficients between experiments with fully attested data and experiments with missing values for both the primary feature (the one with values removed by missingness) and the secondary feature (no data removed). We also calculated the difference in the

regime	strategy	t	p
mcar	listwise_del	0.389	0.697
mcar	mean_imputer	7.684	0.0
mcar	em_imputer	12.336	0.0
mar	listwise_del	27.859	0.0
mar	mean_imputer	28.509	0.0
mar	em_imputer	48.919	0.0
mnar	listwise_del	0.331	0.741
mnar	mean_imputer	9.535	0.0
mnar	em_imputer	36.687	0.0

TABLE 1: Results of pairwise t-tests comparing difference scores for the primary coefficient.

regime	strategy	t	p
mcar	listwise_del	0.005	0.996
mcar	mean_imputer	-2.28	0.023
mcar	em_imputer	-3.745	0.0
mar	listwise_del	-29.256	0.0
mar	mean_imputer	-2.437	0.015
mar	em_imputer	-2.876	0.004
mnar	listwise_del	-3.486	0.0
mnar	mean_imputer	-0.128	0.898
mnar	em_imputer	0.072	0.943

TABLE 2: Results of pairwise t-tests comparing difference scores for the secondary coefficient.

mean squared error of the models between the full datasets and those with missingness applied.

Pairwise independent Welch’s t-tests were performed on differences in the model coefficients and model error between the fully attested data and the three strategies for imputing missing values for each of the three kinds of missingness regimes, for a total of 9 tests. To avoid inflating the overall error rate for each family of comparisons, we used the Bonferroni correction and set the alpha for each individual test to 0.005.

Experiments were completed on a server with an AMD Phenom II X4 955 3.2 GHz processor running Ubuntu 16.04, under Anaconda Python 3.5.4, impute 0.0.4, Numpy 1.13.1, scikit-learn 0.19.0. The code used to run these experiments, the data they generated, a Jupyter notebook containing the code for generating the statistics and plots in this paper, and frozen requirements for the code environment is publicly available at <https://github.com/deniederhut/safe-handling-instructions-for-missing-data>.

Results

Pairwise t-tests conducted on the coefficients of the primary feature show significant differences from zero in seven of the nine cases (Table 1). The only cases where the model learned a similar coefficient involved the use of listwise deletion as a strategy for handling missing data. The smallest difference was observed for cases missing completely at random (stochastically). The largest differences were observed when data were missing at random.

Pairwise t-tests conducted on the difference scores for the secondary coefficient show a similar pattern of results (Table 2). Specifically, the only case in which the estimated parameter for the feature without any missingness applied to it was close to

regime	strategy	t	p
mcAR	listwise_del	-1.332	0.183
mcAR	mean_imputer	-5.643	0.0
mcAR	em_imputer	-7.297	0.0
maR	listwise_del	-46.945	0.0
maR	mean_imputer	-54.322	0.0
maR	em_imputer	-52.646	0.0
mnAR	listwise_del	-9.102	0.0
mnAR	mean_imputer	-12.127	0.0
mnAR	em_imputer	-17.626	0.0

TABLE 3: Results of pairwise t-tests comparing difference scores for the model error.

zero was when data were missing completely at random, and the missing cases were removed listwise. The largest differences in the coefficient for the secondary feature were observed for data missing at random or missing not at random, when the missingness strategy employed was listwise deletion. Listwise deletion tends to cause the coefficient for the secondary feature to be underestimated, while both imputation strategies tend to cause the coefficient to be overestimated.

Pairwise t-tests applied to the overall model error show a similar pattern of results, where the only difference score that is close to zero is for the case of listwise deletion applied to a dataset where values are missing completely at random (Table 3). The largest increases in model error is observed when data are missing at random, no matter which strategy for handling missingness is used.

Discussion

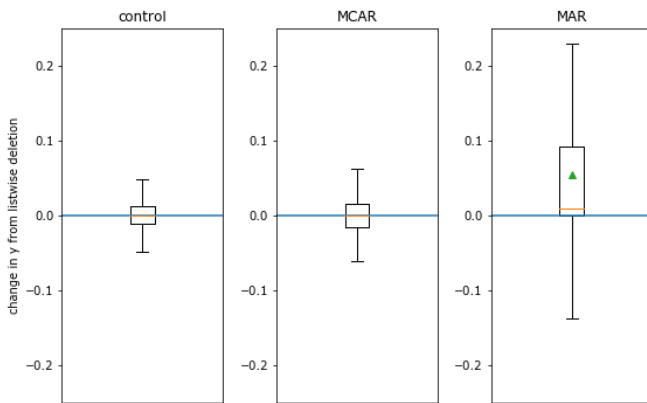


Fig. 2: Changes in the coefficient of y when using listwise deletion across different missingness regimes.

We find that deleting records with missing values is only safe when data are missing completely at random. Under other missingness regimes, this strategy produced biased coefficients for all features, and significantly worse model errors. Interestingly, listwise deletion as a strategy produced the largest bias of all tested strategies in features with no missing data, significantly overestimating their importance to the model (Fig. 2). This suggests that unsafe use of listwise deletion may be one contributing factor in spurious correlations and findings that otherwise fail to replicate.

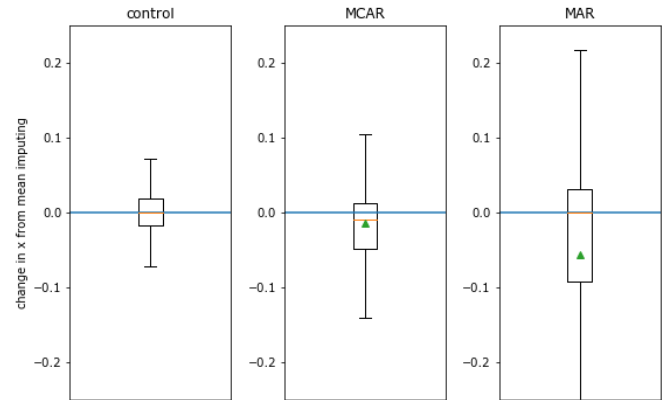


Fig. 3: Changes in the coefficient of x when using single imputation across different missingness regimes.

Single imputation, or using a feature mean or median as replacement for missing data, results in biased coefficients and significantly larger model errors no matter what kind of process created the missingness in the dataset (Fig. 3). As such, it is our recommendation that it not be used. However, in this set of experiments single imputation did produce smaller biases in model features that were not missing any data.

We were surprised by the poor performance of expectation maximization during the experiment given the widespread evidence of its effectiveness in prior literature [shah-et-al-2014]. This discrepancy could be due to a mistake in the design of the experiment, or due to the algorithm’s implementation in `impyle`. As far as we are aware, well-tested multiple imputation libraries like MICE [vanbuuren-groothuisoudshoorn-2011], Amelia [blackwell-honaker-king-2017], and MissForest [stekhoven-buhlmann-2012], have yet to be directly ported to Python⁴.

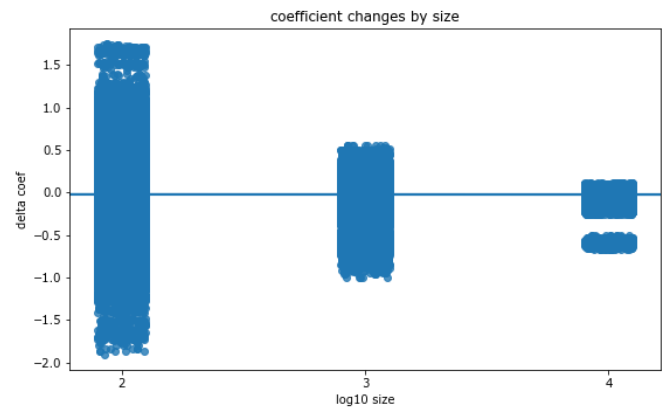


Fig. 4: Changes in the coefficient of x by the size of the total dataset.

As a final comment, we often hear that the solution for missing values is simply to collect more data. However, unless this additional data collection explicitly addresses missingness by correcting the acquisition process (per Inglis), the additional data has the paradoxical effect of making the biases worse. The expected magnitude of the bias does not change with data size—this is governed by the missingness regime and the fraction of missing data. However, the variance in the bias across repeated

experiments will shrink, leading to confidence in the estimated coefficients that is both misplaced and inflated (fig. 4).

Guidelines

We include here guidelines for researchers to use when handling missing data to ensure that it is done safely.

- 1) Try to construct your acquisition step such that there will not be missing values. This may involve following up with individual cases to find why they are non present, so plan to track to provenance of your data.
- 2) In addition to your primary features of interest, collect data that are known to be causally related or correlated. These are called auxiliary features and will help you establish the missingness regime for your data and generate realistic estimates for missing values if needed.
- 3) Once your data have been collected, examine them for patterns of missingness. A common approach is to build a missingness indicator for each feature with missing values, and run pairwise correlations against other features in the dataset. This is more effective with good auxiliary features.
- 4) If you are 100% sure that your missingness is MCAR, you have the option of using listwise deletion, keeping in mind that this should not be done for analyses with low statistical power.
- 5) Otherwise use a modern multiple imputation technique like MICE or MO, and generate 5-10 imputed datasets. Be sure to create any derived features that you plan on including in your final model before the imputation step.
- 6) Run the rest of your analysis as planned for each of the imputed datasets, and report the average parameters of all of the imputed models.
- 7) When you report your results, include the fraction of missing values, the pattern of missing values, and the strategy used to handle them. If your imputed models have widely diverging results, you should report descriptive statistics for any parameters that are highly variable.

Case Study

We can illustrate the use of these guidelines with a real-world case study. The data we'll use is from Scott Cole's open source dataset on burrito quality in San Diego⁵. The dataset consists of approximately 400 ratings of burritos from different restaurants within San Diego, where the ratings for each burrito include five point Likert scores for overall quality, cost, mean, uniformity, salsa, and wrap (the tortilla). The dataset also includes indicator variables for the presence of various ingredients in the burritos, including common ingredients like beans and avocado, and uncommon ones, like sushi and taquitos.

The indicator variables were recoded to work with scikit-learn, and the Likert scores were normalized on a per-rater basis to increase the inter-rater reliability. This brought the dataset down to an effective size of 231 observations. We then used a decision tree (with no hyperparameter tuning) to generate a reference model for predicting overall burrito quality given the individual ratings and presence/absence of ingredients.

The individual ingredients in the burrito don't seem to contribute much to the overall score (Table 4). The quality of the meat emerged as the most important feature in a good burrito, with the

feature	importance
Meat	0.54674656983
Salsa	0.12792116636
Uniformity	0.15980891451

TABLE 4: Features with the highest importance ratings on the fully attested burritos dataset, under a decision tree regressor with no tuning.

quality of the salsa and the uniformity of ingredients throughout the length of the burrito as the next two most important features.

We then impose a regime of MAR on our dataset, removing one ranking score randomly from every record that falls above the 30th percentile for burrito rankings. The causal explanation for this might be something like reviewers are more likely to forget to record data about their burritos when the burrito is tasty, because they are too busy enjoying it.

```
rows = df[df.overall > df.overall.quantile(.3)].index
cols = np.random.choice(['Cost', 'Meat', 'Salsa',
                        'Uniformity'], rows.size)
for row, col in zip(rows, cols):
    df.loc[row, col] = np.nan
```

Because we are using data from another research team, there isn't much we can do with respect to steps 1 and 2 in the guidelines above. So we start with step 3, looking for patterns in the missingness in our dataset, by constructing an indicator for missing values:

```
df['has_nulls'] = pd.isnull(df).sum(axis=1)
```

and then running a correlation against the variables of our dataset (Fig. 5). There is a large correlation ($r=0.8$) between the number of missing values and the overall burrito quality, and moderate correlations ($0.4 < r < 0.6$) with other key rankings, including the quality of the meat and salsa in the burrito.

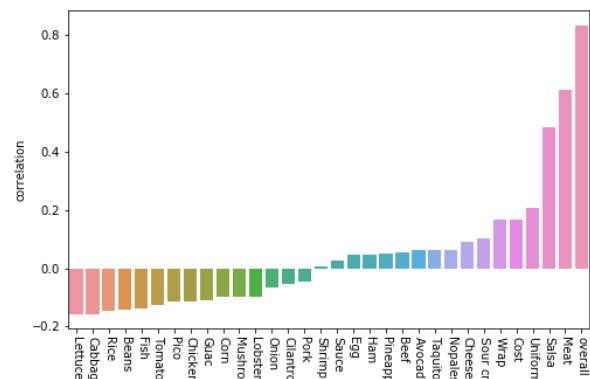


Fig. 5: Pearson correlation strength of model features with count number of missing values per observation.

These correlations indicate that our data are not MCAR, and so we will proceed with multiple imputation. We create five imputed datasets, and train the same untuned decision tree regressor on each of them as above, recording the important features and model scores for each trial. For comparison, we will also run train the model on data using single imputation and listwise deletion.

The multiple imputation dataset returns feature importances that are similar to those found in the model run on the fully attested

feature	importance
Meat	0.42690684148
Salsa	0.14982927778
Uniformity	0.21762993715

TABLE 5: Features from one trial of a dataset using multiple imputation (here, the expectation maximization procedure found in *imp्यूte*).

data, where the meat quality was the most important feature, followed by uniformity and salsa, in that order (Table 5). The single imputation and listwise deletion models both fail to recover the importance of meat quality in the burrito, and compensate for this by overestimating the importance of either the salsa, the uniformity, or the cost.

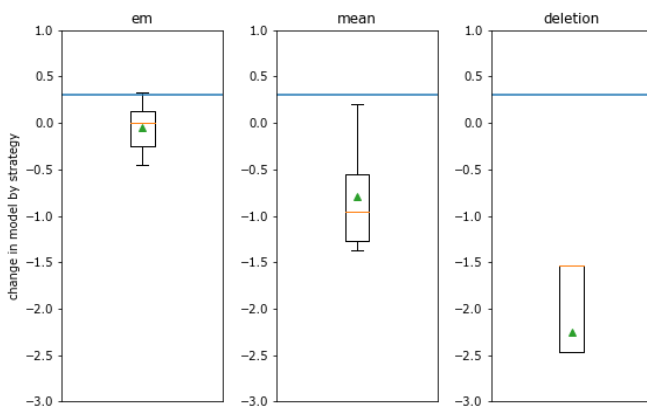


Fig. 6: Distribution of model score for decision trees trained under multiple imputation, single imputation, and listwise deletion. The score obtained on the fully attested model is the reference line in blue.

When comparing model scores (here, the coefficient of determination), none of the models which have had data removed perform as well as the fully attested model (Fig. 6). However, the score on the best model only falls within the range of models trained on multiple imputation data, and not those trained on deleted or singly imputed data. Listwise deletion is the worst performing model here, largely because of the reduced size of the dataset (76 observations).

In our final report, we would include in our methods section that 70% of observations were missing data for at least one feature. We would say that the presence of missing values showed a strong correlation with overall burrito quality, meat quality, and salsa quality, leading us to speculate that people are less likely to fill out surveys when thoroughly relishing a good burrito. We would say that we imputed values using expectation maximization, and that we are reporting averaged results from five separate imputations.

Conclusion

Missing values are a widespread issue in many analytical fields. To handle them safely, there must be some understanding of the kind of process that generated them. Data that are missing completely at random (stochastically) do not create bias during parameter estimation, and can be handled by removing rows with missing values. Missing values that exhibit a definite pattern or dependency

need to be replaced by reasonable estimates using a modern multiple imputation technique. Finally, to ensure reproducibility, statistics and decisions at each of these steps should be reported.

REFERENCES

- [blackwell-honaker-king-2017] M. Blackwell, J. Honaker, and G. King. *A unified approach to measurement error*, *Sociological Methods & Research*, 46:303-341, 2017. doi:10.1177/0049124115585360.
- [vanbuuren-groothuisoudshoorn-2011] S. van Buuren and K. Groothuis-Oudshoorn. *mice: multivariate imputation by chained equations in R*, *Journal of Statistical Software*, 45, 2011.
- [imp्यूte] E. Law. *imp्यूte*, <https://pypi.org/project/imp्यूte/>.
- [little-rubin-2002] R. Little and D. Rubin. *Statistical analysis with missing data (2nd ed.)*. New York, NY: Wiley, 2002. doi:10.1002/9781119013563.
- [newman-2014] D. Newman. *Missing data: five practical guidelines*, *Organizational Research Methods*, 17:372-411. doi:10.1177/1094428114548590.
- [perkins-et-al-2018] N. Perkins, S. Cole, O. Harel, E. Tchetgen, B. Sun, E. Mitchell, and E. Schisterman. *Principled approaches to missing data in epidemiological studies*. *American Journal of Epidemiology*, 187:568-575, 2018. doi:10.1093/aje/kwx348.
- [rubin-1976] D. Rubin. *Inference and missing data*, *Biometrika*, 63:581-592, 1976. doi:10.1093/biomet/63.3.581.
- [schafer-graham-2002] J. Schafer and J. Graham. *Missing data: Our view of the state of the art*. *Psychological Methods*, 7:147-177, 2002. doi:10.1037/1082-989X.7.2.147.
- [shah-et-al-2014] A. Shah, J. Bartlett, J. Carpenter, O. Nicholas, and H. Hemingway. *Comparison of random forest and parametric imputation models for imputing missing data using MICE: A CALIBER study*. *American Journal of Epidemiology*, 179:764-774, 2014. doi:10.1093/aje/kwt312.
- [stekhoven-buhlmann-2012] D. Stekhoven and P. Bühlmann. *MissForest - non-parametric missing value imputation for mixed-type data*, *Bioinformatics*, 28:112-118, 2012. doi:10.1093/bioinformatics/btr597.
- [sullivan-et-al-2017] T. Sullivan, L. Yelland, K. Lee, P. Ryan, and A. Salter. *Treatment of missing data in follow-up studies of randomised controlled trials: A systematic review of the literature*. *Clinical Trials*, 14:387-395, 2017. doi:10.1177/1740774517703319.
1. Named after Dr. Ben Inglis of the University of California, the Inglis Conjecture states that it requires less effort to fix the data acquisition step than to perform post hoc statistical corrections.
 2. *Random* in the sense of a random variable, which is a statistical designation roughly corresponding to a dependent variable.
 3. An auxiliary feature is one which measures a related variable but is not necessarily included in the final model.
 4. *imp्यूte* has an imputing function called MICE, but implements a modification of the original algorithm.
 5. Licensed under MIT, and available at <https://github.com/srcole/burritos>. You can watch Scott's lightning talk about this dataset from SciPy 2017 at https://youtu.be/f-Vcq_anPaY?t=47m44s.

Text and data mining scientific articles with allofplos

Elizabeth Seiver*, M Pacer[§], Sebastian Bassi[‡]

Abstract—Mining scientific articles is hard when many of them are inaccessible behind paywalls. The Public Library of Science (PLOS) is a non-profit Open Access science publisher of the single largest journal (*PLOS ONE*), whose articles are all freely available to read and re-use. `allofplos` is a Python package for maintaining a constantly growing collection of PLOS's 230,000+ articles. It also efficiently parses these article files into Python data structures. This article will cover how `allofplos` keeps your articles up-to-date, and how to use it to easily access common article metadata and fuel your meta-research, with actual use cases from inside PLOS.

Index Terms—Text and data mining, metascience, open access, science publishing, scientific articles, XML

Introduction

Why mine scientific articles?

Scientific articles are the standard mechanism of communication in science. They embody a clear way by which human minds across centuries and continents are able to communicate with one another, growing the total sum of knowledge. Scientific articles are unique resources, in that they are the material artifacts by which this cultural exchange is made concrete and persistent. They offer a unique source of insight into the history of carefully argued, hard-won knowledge. Accordingly because they are made of annotated text, they offer unique opportunities for well-defined text and data mining problems. Importantly, because PLOS represents the largest single journal in the history of publishing, it has collected an excellent corpus for this study, spanning seven journals that specialize in biology and medicine. Equally importantly, because PLOS is Open Access, the opportunity to use this data set is available to anyone capable of downloading and analyzing it. The `allofplos` library enables more people to do that more easily.

What is `allofplos`?

`allofplos` is a Python package for downloading and maintaining up-to-date scientific article corpora, as well as parsing PLOS XML articles in the JATS (Journal Article Tag Suite) [`jats`] format. It is available on PyPI [`allofplospypi`] as well as a GitHub repository [`allofplogh`]. Many existing Python packages for parsing XML and/or JATS focus on defensive parsing, where the structure is assumed not to be reliable or the document is immediately

converted to another intermediate format (often JSON) and XML is just a temporary stepping stone. `allofplos` uses `lxml` [`lxml05`], which is compiled in C, for fast XML parsing and conversion to familiar Python data structures like lists, dictionaries, and datetime objects. The intended audience is researchers who are familiar with scientific articles and Python, but may not be familiar with JATS XML. Other related tools include a parser from fellow Open Access publisher eLife [`elife`] as well as the Open Access subset for downloading OA articles in bulk from PubMed Commons (PMC) [`pmc`].

Functionality

The primary function of `allofplos` is to download and maintain a corpus of PLOS articles. To enable users to parse articles without downloading 230,000 XML files, `allofplos` ships with a starter directory of 122 articles (`starterdir`), and includes commands for downloading a 10,000 article demo corpus as well. The default path to a corpus is stored as the variable `corpusdir` in the Python program, and first checks for the environment variable `$PLOS_CORPUS` which overrides that default location. If you have used pip to install the program, specifying `$PLOS_CORPUS` will ensure that the article data will not be overwritten when you update the `allofplos` package, as the default location is within the package. (Forking/cloning the GitHub repository avoids this problem, because the default corpus location is in the `.gitignore` file.)

```
import os
os.environ['PLOS_CORPUS'] = 'path/to/corpus_directory'
from allofplos import update
update.main()
```

Downloading new articles can also be accessed via the command line:

```
$ export PLOS_CORPUS="path/to/corpus_directory"
$ python -m allofplos.update
```

If no articles are found at the specified corpus location, it will initiate a download of the full corpus. This is a 4.6 GB zip file stored on Google Drive, updated daily via an internal PLOS server, that then is unzipped in that location to around 25 GB of 230,000+ XML articles. For incremental updates of the corpus, `allofplos` first scans the corpus directory for all DOIs (Digital Object Identifiers) [`doi`] of all articles (constructed from filenames) and compares that with every article DOI from the PLOS search API. The missing articles are then downloaded individually in a rate-limited fashion from links that are constructed using the DOIs. Those files are identical to the ones in the `.zip` file. The `.zip` file prevents users from needing to scrape the entire PLOS website for the XML files, and "smartly" scrapes only the latest articles. For a subset of

* Corresponding author: elizabeth.seiver@gmail.com

[§] Netflix

[‡] Globant

provisional articles called "uncorrected proofs", it checks whether the final version is available, and downloads the updated version if so. The files are then ready for parsing and analysis.

Article corpora and parsing

To initialize a corpus (defaults to `corpusdir`, or the location set by the `$PLOS_CORPUS` environmental variable), use the `Corpus` class. This points `alloplos` at the directory of articles to be analyzed.

```
from alloplos import Corpus
corpus = Corpus()
```

To analyze the starter directory, also import `starterdir` and set `corpus = Corpus(starterdir)`. The number of articles in the corpus can be found with `len(corpus)`. The list of every DOI for every article in the corpus can be found at `corpus.dois`, and the path to every XML file in the corpus directory at `corpus_filenames`. To select a random `Article` object, use `corpus.random_article`. To select a random list of ten `Article` objects, use `corpus.random_sample(10)`. You can also iterate through articles as such:

```
for article in corpus[:10]:
    print(article.title)
```

Because DOIs contain semantic meaning and XML filenames are based on the DOI, if you systematically loop through the corpus, it will not be a representative sample but rather will implicitly progress first by journal name and then by publication date. The iterator for `Corpus()` puts the articles in a random order to avoid this problem.

The `Article` class

As mentioned above, you can use the `Corpus` class to initialize an `Article()` object without calling `Article` directly. An `Article` takes a DOI and the location of the corpus directory to read the accompanying XML document into `lxml`.

```
art = Article('10.1371/journal.pcbi.1004692')
```

The `lxml` tree of the article is memoized in `art.tree` so it can be repeatedly called without needing to re-read the XML file.

```
>>> type(art.tree)
lxml.etree._ElementTree
```

Article parsing in `alloplos` focuses on metadata (e.g., article title, author names and institutions, date of publication, Creative Commons copyright license [cc], JATS version/DTD), which are conveniently located in the front section of the XML. We designed the parsing API to quickly locate and parse XML elements as properties without needing to know the JATS tagging format.

```
>>> art.doi
'10.1371/journal.pcbi.1004692'
>>> art.title
'Ensemble Tractography'
>>> art.journal
'PLOS Computational Biology'
>>> art.pubdate
datetime.datetime(2016, 2, 4, 0, 0)
>>> art.license
{'license': 'CC-BY 4.0',
 'license_link':
  'https://creativecommons.org/licenses/by/4.0/',
 'copyright_holder': 'Takemura et al',
```

```
'copyright_year': 2016}
>>> art.dtd
'JATS 1.1d3'
```

For author information, `Article` reconciles and combines data from multiple elements within the article into a clean standard form, including author email addresses and affiliated institutions. Property names match XML tags whenever possible.

Using XPath

While the `Article` class handles most basic metadata within the XML files, users may also wish to analyze the content of the article more directly. The XPath query language is built into `lxml` and provides a way to search for particular XML tags or attributes. (Note that XPath will always return a list of results, as element tags and locations are not unique.) You can perform XPath searches on `art.tree`, which also works well for finding article elements that are not `Article` class properties, such as the acknowledgments, which have the tag `<ack>`.

```
>>> acknowledge = art.tree.xpath('//ack/p')[0]
>>> acknowledge.text[:41]
'We thank Ariel Rokem and Jason D. Yeatman'
```

For users who are more familiar with XML or want to perform quality control checks on XML files, XPath searches can find articles that match a particular XML structure. For example, PLOS's production team needed to find articles that had a `<list>` item anywhere within a `<boxed-text>` element. They iterated through the corpus using `art.tree.xpath('//boxed-text//list')`.

Use case: searching Methods sections

We can put these pieces together to make a list of articles that use PCR (Polymerase Chain Reaction, a common molecular biology technique) in their Methods section (`pcr_list`). The body of an article is divided into sections (with the element tag `<sec>`) and the element attributes of Methods sections are either `{'sec-type': 'materials|methods'}` or `{'sec-type': 'methods'}`. In addition to importing `alloplos`, the `lxml.etree` module needs to be imported to turn XML elements into Python strings via the `tostring()` method.

```
import lxml.etree as et
pcr_list = []
for article in corpus.random_sample(20):

    # Step 1: find Method sections
    methods_sections = article.root.xpath(
        "//sec[@sec-type='materials|methods']")
    if not methods_sections:
        methods_sections = article.root.xpath(
            "//sec[@sec-type='methods']")

    for sec in methods_sections:

        # Step 2: turn the method sections into strings
        method_string = et.tostring(sec, method='text',
            encoding='unicode')

        # Step 3: add DOI if 'PCR' in string
        if 'PCR' in method_string:
            pcr_list.append(article.doi)
            break
        else:
            pass
```


Included SQLite database

The *allofplos* code includes a SQLite database with all articles in starter directory. In this release there are 122 records that represents a wide range of papers. In order to use the database, the user needs a SQLite client. The official client is command line based and can be downloaded from <https://www.sqlite.org/download.html>. The database can also be displayed on graphical viewers such as [DB Browser for SQLite](#) and [SQLiteStudio](#). There are also some options to query the database online, without installing any software, like <https://sqliteonline.com/> and <http://inloop.github.io/sqlite-viewer/>.

The main table of the database is *plosarticle*. It has the DOI, title, abstract, publication date and other fields that link to other child tables, like *articletype* and *journal_id*. The corresponding author information is stored in the *correspondingauthor* table and is linked to the *plosarticle* table using the relation table called *coauthorplosarticle*.

For example, to get all papers whose corresponding authors are from France:

```
SELECT DOI FROM plosarticle
JOIN coauthorplosarticle ON
coauthorplosarticle.article_id = plosarticle.id
JOIN correspondingauthor ON
(correspondingauthor.id =
coauthorplosarticle.corr_author_id)
JOIN country ON
country.id = correspondingauthor.country_id
WHERE country.country = 'France';
```

This will return the DOIs from three papers from the starter database:

```
10.1371/journal.pcbi.1004152
10.1371/journal.ppat.1000105
10.1371/journal.pgen.1002912
10.1371/journal.pcbi.1004082
```

The researcher can avoid using SQL queries by using the included Object-relational mapping (ORM) models. The ORM library used is *peewee*. A file with sample queries is stored in the repository with the name of *allofplos/dbtoorm.py*. Part of this file defines all Python classes that corresponds to the SQLite Database. These class definitions are from the beginning of the file until the comment marked as `# End of ORM classes creation`.

After this comment, there is an example of how to build a query. The following query is the *peewee* compatible syntax that constructs the same SQL query as outlined before:

```
query = (Plosarticle
    .select()
    .join(Coauthorplosarticle)
    .join(Correspondingauthor)
    .join(Country)
    .join(Journal,
        on=(Plosarticle.journal == Journal.id))
    .where(Country.country == 'France'))
```

This will return a *query* object. This object can be walked over with a for loop as any Python iterable:

```
for papers in query:
    print(papers.doi)
```

SQLite database constructor

There is a script at *allofplos/makedb.py* that can be used to generate the SQLite Database from a directory full of XML

articles. This script was used to generate the included **starter.db**. If the user wants to make another version, from another subset (or from the whole corpus), this script will be useful.

To generate a SQLite DB with all the files currently in the *Corpus* directory, and save the DB as *mydb.db*:

```
$ python makedb.py --db mydb.db
```

There is an option to generate a DB with only a random subset of articles. For a DB with 500 articles randomly selected, use:

```
$ python makedb.py --random 500 --db mydb.db
```

Future directions

We also have plans for future updates to *allofplos*. First, we plan to make the article parsing publisher-neutral, allowing for reading JATS content from other publishers in addition to PLOS. Second, we want to improve incremental corpus updates so that all changes can be downloaded and updated via a standardized mechanism such as a hash table. This includes 'silent republications', where articles are updated online without an official correction notice (the substance of the article is unchanged, but the XML has been updated). While the local *allofplos* server has methods for catching these changes and updating the zip file appropriately, there is not currently a way to make sure a user's local corpus copy reflects all of those changes. Third, we want to expand the possibilities of multiple corpora and allow for article versioning, such as for comparing older and newer versions of articles instead of just replacing them entirely. And finally, we want to expand and integrate the functionality of the *sqlite* database so that selecting a subset of articles based on metadata criteria such as journal, publication date, or author is faster and easier than looping through each XML file individually.

Conclusions

As more scientific articles are published, it will become more important that these articles can be analyzed in aggregate. Tools like *allofplos* make such an effort much easier. With an intuitive and straightforward *Corpus()* and *Article()* APIs, *allofplos* avoids much of the complexity of parsing xml for new users, while still enabling XML experts the flexibility and power needed to accomplish their aims. By building in the ability to automatically update and maintain the corpus, people can trust that they have the most state-of-the-art data without needing to manually check the >230,000 articles (a task few would undertake). By connecting this information to database technologies, *allofplos* enables quickly accessing data when that efficient access is needed. By making strides in all of these directions *allofplos* demonstrates itself to be a valuable tool in the scientific python toolkit.

REFERENCES

- [lxml05] Behnel, S., Faassen, M. et al. (2005), lxml: XML and HTML with Python, <http://lxml.de>.
- [cc] Creative Commons Licenses. <https://creativecommons.org/licenses/>
- [allofplogh] *allofplos* GitHub repository. <https://github.com/PLOS/allofplos>
- [allofplospypi] *allofplos* PyPI repository. <https://pypi.org/project/allofplos/>
- [jats] JATS NIH/NISO standard. <https://jats.nlm.nih.gov/publishing/tag-library/1.1d3/chapter/how-to-read.html>
- [elife] *elife-tools* GitHub repository. <https://github.com/elifesciences/elife-tools>

[doi] Digital Object Identifiers. https://www.doi.org/doi_handbook/1_Introduction.html
[pmc] PMC Open Access Subset. <https://www.ncbi.nlm.nih.gov/pmc/tools/openftlist/>

Sparse: A more modern sparse array library

Hameer Abbasi^{‡*}

<https://youtu.be/xH5eVcb1S1A>

Abstract—This paper is about sparse multi-dimensional arrays in Python. We discuss their applications, layouts, and current implementations in the SciPy ecosystem along with strengths and weaknesses. We then introduce a new package for sparse arrays that builds on the legacy of the `scipy.sparse` implementation, but supports more modern interfaces, dimensions greater than two, and improved integration with newer array packages, like XArray and Dask. We end with performance benchmarks and notes on future work. Additionally, this work provides a concrete implementation of the recent NumPy array protocols to build generic array interfaces for improved interoperability, and so may be useful for broader community discussion.

Index Terms—sparse, sparse arrays, sparse matrices, `scipy.sparse`, `ndarray`, `ndarray` interface

Introduction

Sparse arrays are important in many situations and offer both speed and memory benefits over regular arrays when solving a broad spectrum of problems. For example, they can be used in solving systems of equations [LN89], solving partial differential equations [MR91], machine learning problems involving Bayesian models [Tip01] and natural language processing [NTK11].

As a motivating example, consider two NumPy arrays with a shape of $(10 \times 5, 10 \times 5)$ and only five nonzero elements per row. Computations on such arrays, such as addition, multiplication and so on would perform the operation on each of the 10^{10} elements individually, taking up a large amount of time and memory.

If we instead focused on just the nonzero elements in each array and worked with those, we would be down to at most 10^6 elements to work with, a *huge* improvement. If we were smart about how the array would be stored, we could also bring down memory usage as well. This is, in essence, what sparse arrays do and what they're used for.

Traditionally, within the SciPy ecosystem, sparse arrays have been provided within SciPy [Sci18] in the submodule `scipy.sparse`, which is arguably the most feature-complete implementation of sparse matrices within the ecosystem, providing support for basic arithmetic, linear algebra and graph theoretic algorithms.

However, it lacks certain features which prevent it from working nicely with other packages in the ecosystem which consume NumPy's [Num18] `ndarray` interface:

- It doesn't follow the `ndarray` interface (rather, it follows NumPy's deprecated `matrix` interface)
- It is limited to two dimensions only (even one-dimensional structures aren't supported)

In addition, `scipy.sparse` is depended on by many downstream projects, which makes removing NumPy's `matrix` interface that much more difficult, and limits usage of both `ndarray` style duck arrays and `scipy.sparse` arrays within the same codebase.

This is important for a number of other packages that are quite innovative, but cannot take advantage of `scipy.sparse` for these reasons, because they expect objects following the `ndarray` interface. These include packages like Dask [Das18] (which is useful for parallel computing, even across clusters, for both NumPy arrays and Pandas dataframes) and XArray [xar18] (which extends Pandas dataframes to multiple dimensions).

Both of these frameworks could benefit tremendously from sparse structures. In the case of Dask, it could be used in combination with sparse structures to scale up computational tasks that need sparse structures. In the case of XArray, datasets with large amounts of missing data could be represented efficiently, as well as other benefits such as broadcasting by axis name rather than by rather opaque axis positions.

In this paper, we present Sparse [Spa18], a sparse array library that supports arbitrary dimension sparse arrays and supports most common parts of the `ndarray` interface. It supports basic arithmetic, application of `ufunc`s directly to sparse arrays (including with broadcasting), most common reductions, indexing, concatenation, stacking, transpose, reshape and a number of other features. The primary format in this library is based on the coordinate format, which stores indices where the array is nonzero, and the corresponding data.

Since a full explanation of usage would be a repeat of the NumPy user manual and the package documentation, we move on to some of the design decisions that went into making this package, including some challenges we had to face some optimizations, applications and possible future work.

Algorithms and Challenges

Choice of storage format

We chose the COO format for its simplicity while storing and accessing elements, even though it isn't the most efficient storage format. In this format, two dense arrays are required to store the sparse array's data. The first is a coordinates array, which stores the coordinates where the array is nonzero. This array has a shape $(\text{ndim}, \text{nnz})$. The second is a data array, which stores the

* Corresponding author: hameerabbasi@yahoo.com

‡ TU Darmstadt

dim1	dim2	dim3	...	data
0	0	0	...	10
0	0	3	...	13
0	2	2	...	9
...
3	1	4	...	21

TABLE 1
A visual representation of the COO format.

data corresponding to each coordinate, and thus it has the shape $(nnz,)$. Here, $ndim$ represents the number of dimensions of the array and nnz represents the number of nonzero entries in the array.

For simplicity of operations in many cases, the coordinates are always stored in C-contiguous order. Table 1 shows a visual representation of how data is stored in the COO format.

We use whatever data-type the source array has for the data array and `np.int64` for the coordinates array. This means that, assuming $ndim = 3$ and $dtype.itemsize = 8$ (as is the case for a data type of `np.int64`, `np.uint64` and `np.float64`), the tipping point versus dense arrays for memory usage will be a density of 0.25, with the benefit increasing with the inverse of the density.

Element-wise operations

Element-wise operations are an important and common part of any array interface. For example, arithmetic, casting an array, and all NumPy `ufunc`s are common examples of element-wise operations.

These turn out to be simple for NumPy arrays, but are surprisingly complex for sparse arrays. The first problem to overcome was that there was no dependency on Numba [Ana18]/Cython [Cyt18]/C++ at the time that this algorithm was to be implemented, and a discussion was ongoing about which algorithm to use. [Spae] I, therefore wished to solve the problem in pure NumPy, therefore looping over all possible nonzero coordinates was not an option, and we had to process the coordinates and data in batches. The batches that made sense at the time were something like the following:

- 1) Coordinates in the first array but not in the second.
- 2) Coordinates in the second array but not in the first.
- 3) Coordinates in both arrays simultaneously.

This algorithm (when applied to multiple inputs instead of just two) looks like the following:

```
all_coords = []
all_data = []

for each combination of inputs where some are zero
and some nonzero:
    if all inputs are zero:
        continue

    coords = find coordinates common to
              nonzero inputs
    coords = filter out coordinates that are
              in zero inputs
    data = apply function to data corresponding
           to these coordinates

    all_coords.append(coords)
```

```
all_data.append(data)
```

concatenate `all_coords` and `all_data`

The addition of broadcasting makes this problem even more complex to solve, as it turns out that for sparse arrays, simply broadcasting all arrays to a common shape and then performing element-wise operations is not the most efficient way to perform such an operation.

Consider two arrays, one shaped $(n,)$ and another shaped (m, n) , both with only one nonzero entry. If all we wanted to do was multiply them, the result would have just one nonzero entry, yet broadcasting the first array would result in an array with m nonzero entries (which clearly isn't the most optimal way to do things). For this reason, we chose to handle broadcasting within the algorithm itself, instead of broadcasting all inputs upfront.

Effectively, this resulted in the following algorithm, which doesn't have the limitation mentioned above. This is because any zeros are filtered out before any broadcasting is done:

```
all_coords = []
all_data = []

for each combination of inputs where some are zero
and some nonzero:
    if all inputs are zero:
        continue

    coords = find coordinates common to
              nonzero inputs
              (for dimensions that are not being
              broadcast in both, with repetition
              similar to an SQL outer join)
    data = apply function to data corresponding
           to these coordinates

    coords, data = filter out zeros from coords/data

    coords, data = filter out coordinates/data that
                  are in zero inputs
                  (again, for non-broadcast dimensions)

    broadcast coordinates and data to output shape

    all_coords.append(coords)
    all_data.append(data)

concatenate all_coords and all_data
```

The full implementation can be found in [Spaa]. While this algorithm is effective at applying all sorts of element-wise operations for any amount of inputs, it does have a few drawbacks:

- It's slower than `scipy.sparse`, because
 - It loops over all possible combinations of zero/nonzero coordinates, which makes it $O((2^{nin} - 1) \times nnz)$ in the worst case, where nin is the number of inputs to the operation and nnz are the number of nonzero elements.
 - It's in COO format rather than CSR/CSC.
 - `scipy.sparse` uses specialized code paths for each operation that greatly reduce the strain on the CPU whereas we keep everything generic.
- In the current implementation, sorting of coordinates is sometimes done unnecessarily.

This can be improved in the future in the following ways:

- Looping over possibly nonzero coordinates with something like Numba or Cython.

- This approach will solve most of the speed issues.
 - Sorting will be rendered unnecessary.
 - Specialized code paths introduce a large maintenance burden, but can be implemented.
- Introducing multidimensional CSR/CSC.

You can see the current performance of the code in Table 2.

Currently, the implementation raises a `ValueError` if `ndarray`s are mixed with sparse arrays, or if the operation produces a dense array, such as operations like $y = x + 5$ where x is sparse. This is an intentional design choice: We raise an error to show that the result is likely dense, and that if the user wishes to perform a dense operation, they should convert all arrays involved to dense ones and repeat the operation. This is better than an undesired performance degradation, which can be hard to detect.

However, work is being done to reduce the amount of such errors. For example, there is a feature planned to allow mixed `ndarray`-sparse operations if such operations do not produce dense results e.g. multiplication. [Spac]. Also, we are planning to allow arbitrary fill values in arrays, which will allow for operations such as $y = x + 5$ (if `x.fill_value` was zero, `y.fill_value` will be five). [Spad]

Reductions

We implemented reductions by the elegant concept of a "grouped reduce". The idea is to first group the coordinates by the non-selected axes, and then reduce along the selected axes. This is simple to implement in practice, and also works quite well. Here is some psuedocode that we use for reductions:

```
x = x.transpose((selected_axes, non_selected_axes))
x = x.reshape((selected_axes_size,
              non_selected_axes_size))

y, counts = perform a reduce on x
            grouped by the first coordinate
            using ufunc.reduceat
where counts < non_selected_axes_size, reduce
            an extra time by zero

y = y.reshape(non_selected_axes_shape)
```

The full implementation can be found at [Spab]. Only some reductions are possible with this algorithm at the moment, but most common ones are supported. Supported reductions must have a few properties:

- They must be implemented in the form of `ufunc.reduce`
- The `ufunc` must be reorderable
- Reducing by multiple zeros shouldn't change the result
- An all-zero reduction must produce a zero.

Although these criteria seem restricting, in practice most reductions such as `sum`, `prod`, `min`, `max`, `any` and `all` actually fall within the class of supported reductions. We used `__array_ufunc__` protocol to allow application of `ufunc` reductions to COO arrays. Notable unsupported reductions are `argmin` and `argmax`, because they cannot be implemented in the form `ufunc.reduce`.

This is nearly as fast as the reductions in `scipy.sparse` when reducing along C-contiguous axes, but is slow otherwise. Performance results can be seen in Table 2. Profiling reveals that most of the time in the slow case is taken up by sorting, as

`ufunc.reduceat` expects all "groups" to be right next to each other. This can be improved in the following ways:

- Implement a radix argsort, which will significantly speed up the sorting.
- Perform a "grouped reduce" by other methods, such as how Pandas does it, perhaps by using a `dict` to maintain the results.

Indexing

For indexing, we realize that to construct the new coordinates and data, we can perform two kinds of filtering as to which coordinates will be in the new array and which ones won't.

The first is where we look at the coordinates directly, and then filter them out successively for each given index. For integers, we check for coordinates that are exactly equal to that index. For slices, we similarly check for matching coordinates. We do this for each index. This turns out to be $O(\text{ndim} \times \text{nnz})$ in total, where `ndim` is the number of dimensions of the array to the operation and `nnz` are the number of nonzero elements.

This has a few benefits: it is simple to do and the performance only depends on the size of the input array.

The second is where we look at each integer index in series, and then look at *sub-arrays* for each integer index. Since the coordinates are sorted in lexicographical order, we will have to do a binary search for the start and end of each sub array, and repeat this for each integer index within the previous sub-array. Getting a single item or an integer slice in this case is $O(\text{nidx} \times \log \text{nnz})$. Here, `nidx` is the number of provided integer indices. For slices, we will loop over each possible integer in the slice and repeat the above procedure.

For integer indexing, the second method is almost always faster. For slices, the situation becomes more complicated. Even for slices, in some cases, it is faster to use the second procedure. This happens for small slices, e.g. `x[:10]`.

For other cases, it's wise to initially use the second procedure (to filter out some sub-arrays), and then switch to the first. For example, for `x[:500, :500, :500]`, as using just the second procedure will require a large amount of binary searches (500^3 in this case).

So we used a hybrid approach where the second method is used until there are a sufficiently low number of coordinates left for filtering, then we fall back to simple filtering. Where we do the switch is determined by a heuristic: will the expected number of binary searches be faster in a specific case, or directly filtering the number of left-over coordinates? The overall algorithm is implemented in Numba, because when this algorithm was implemented, the discussion in [Spae] had been resolved. However, it has since been reopened due to further missing features in Numba.

After getting the required coordinates and corresponding data, we apply some simple transformations to it to get the output coordinates and data.

However, one thing is important to realize: indexing sparse arrays is more expensive than indexing dense arrays. Indexes of dense arrays produce a view for any combination of slices and integers, and take $O(\text{nidx})$ time in every case. Sparse arrays take more time, and it's usually not possible to produce a view of the original array.

Benchmark	Sparse	SciPy Sparse	NumPy
Addition	50.8 ms \pm 3.45 ms	2.49 ms \pm 211 μ s	507 ms \pm 6.43 ms
Multiplication	10.7 ms \pm 526 μ s	14.9 ms \pm 1.68 ms	529 ms \pm 13.5 ms
Sum, Axis=0	12 ms \pm 116 μ s	545 μ s \pm 49.8 μ s	97.8 ms \pm 4.19 ms
Sum, Axis=1	959 μ s \pm 23.7 μ s	641 μ s \pm 83.9 μ s	62.7 ms \pm 4.86 ms

TABLE 2

Performance benchmarks comparing Sparse to SciPy and dense NumPy code

Transposing and Reshaping

Transposing corresponds to a simple reordering of the dimensions in the coordinates, along with a re-sorting of the coordinates and data to make the coordinates C-contiguous again.

Reshaping corresponds to linearizing the coordinates and then doing the reverse for the new shape, similar to `np.ravel_multi_index` and `np.unravel_index`. However, we write our own custom implementation for this.

Matrix and tensor multiplication

For `tensordot`, we currently just use the NumPy implementation, replacing `np.dot` with `scipy.sparse.csr_matrix.dot`. This is mainly just transposing and reshaping the matrix into 2-D, using `np.dot` (or `scipy.sparse.csr_matrix.dot` in our case), and performing the reshape and transpose operations in reverse.

For `sparse.dot`, we simply dispatch to `tensordot`, providing the appropriate axes.

This may not always produce a sparse array as output. If we think of each element of the output matrix as a dot product of the appropriate row of the first matrix and the appropriate column of the second matrix, we realize that it may be difficult to guarantee that this will be zero. Indeed, in general, $nnz_{out} \leq nnz_{in1} \times nnz_{in2}$, without knowing much about the structure of the matrix. For some inputs however, the outputs will be relatively sparse (for example for identity matrices and diagonal matrices).

Benchmarks

Because of our desire for clean and generic code as well as using mainly pure Python as opposed to Cython/C/C++ in most places, our code is not as fast as `scipy.sparse.csr_matrix`. It, however, does beat `numpy.ndarray`, provided the sparsity of the array is small enough. The benchmarks were performed on a laptop with a Core i7-3537U processor and 16 GB of memory. Any arrays used had a shape of (10000, 10000) with a density of 0.001. The results are tabulated in Table 2.

The NumPy results are given only for comparison, and for the purposes of illustrating that using sparse arrays does, indeed, have benefits over using dense arrays when the density of the sparse array is sufficiently low.

Outlook and Future Work

We discussed the current leading solution for sparse arrays in the ecosystem, `scipy.sparse`, along with its shortcomings and limitations. We then introduced a new package for N-dimensional sparse arrays, and how it has the potential to address these

shortcomings. We discuss its current implementation, including the algorithms used in some of the different operations and the limitations and drawbacks of each algorithm. We also discuss future improvements that could be made to improve these algorithms.

There are a number of areas we would like to focus on in the future. These include, in very broad terms:

- Better performance
- Better integration with community packages, such as scikit-learn, Dask and XArray
- Support for more of the `ndarray` interface (particularly through protocols)
- Implementation of more linear algebra routines, such as `eig`, `svd`, and `solve`
- Implementation of more sparse storage formats, such as a generalization of CSR/CSC

REFERENCES

- [Ana18] Anaconda, Inc. Numba, 2018. URL: <https://numba.pydata.org/>.
- [Cyt18] Cython developers. Cython, 2018. URL: <http://cython.org/>.
- [Das18] Dask core developers. Dask, 2018. URL: <https://dask.pydata.org/en/latest/>.
- [LN89] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.
- [MR91] Mo Mu and John R Rice. An organization of sparse Gauss elimination for solving PDEs on distributed memory machines. 1991.
- [NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, volume 11, pages 809–816, 2011.
- [Num18] NumPy developers. Numpy, 2018. URL: <https://www.numpy.org/>.
- [Sci18] SciPy developers. Scipy, 2018. URL: <https://www.scipy.org/>.
- [Spaa] Sparse developers. Sparse Implementation: Elementwise. URL: <https://github.com/pydata/sparse/blob/b51d74924d62ff6537b15ce4e1dd4e56080a3b6f/sparse/coo/umath.py#L12>.
- [Spab] Sparse developers. Sparse Implementation: Reductions. URL: <https://github.com/pydata/sparse/blob/b51d74924d62ff6537b15ce4e1dd4e56080a3b6f/sparse/coo/core.py#L564>.
- [Spac] Sparse developers. Sparse Issue: Allow ndarray in elemwise again. URL: <https://github.com/pydata/sparse/issues/124>.
- [Spad] Sparse developers. Sparse Issue: Arbitrary fill value. URL: <https://github.com/pydata/sparse/issues/143>.
- [Spae] Sparse developers. Sparse Issue: Use Cython, Numba, or C/C++ for algorithmic code. URL: <https://github.com/pydata/sparse/issues/143>.
- [Spa18] Sparse developers. Sparse, 2018. URL: <https://sparse.pydata.org/en/latest/>.
- [Tip01] Michael E Tipping. Sparse Bayesian learning and the relevance vector machine. *Journal of machine learning research*, 1(Jun):211–244, 2001.
- [xar18] xarray Developers. xarray, 2018. URL: <https://xarray.pydata.org/en/stable/>.

Bringing ipywidgets Support to plotly.py

Jon Mease^{‡*}

<https://youtu.be/1ndo6C1KWjI>

Abstract—Plotly.js is a declarative JavaScript data visualization library built on D3 and WebGL that supports a wide range of statistical, scientific, financial, geographic, and 3-dimensional visualizations. Support for creating Plotly.js visualizations from Python is provided by the plotly.py library. Version 3 of plotly.py integrates ipywidgets support, providing a host of benefits to plotly.py users working in the Jupyter notebook. This paper describes the architecture of this new version of plotly.py, and presents examples of several of these benefits.

Index Terms—ipywidgets, plotly, jupyter, visualization

Introduction

The Jupyter Notebook [KRKP⁺16] has emerged as the dominant interface for exploratory data analysis and visualization in the Python data science ecosystem. The ipywidgets library [GFC] provides a suite of interactive widgets for use in the Jupyter Notebook, and it serves as a foundation for library authors to build on to create their own custom widgets.

This paper describes our work to bring ipywidgets support to plotly.py version 3. Compared to version 2, plotly.py version 3 brings plotly.py users working in the Jupyter Notebook a host of benefits. Figures already displayed in the notebook may now be updated in-place using property assignment syntax. All properties throughout the entire figure hierarchy are now discoverable using tab completion and documented with informative docstrings. Property values are now fully validated by the Python library and helpful error messages are raised on validation failures. Figure transitions may now be animated. Numpy arrays are now transferred between the Python and JavaScript libraries using a binary serialization protocol for improved performance. Finally, Python callbacks may now be registered for execution upon zoom, pan, click, hover, and data selection events.

Plotly.js Overview

Plotly.js is a JavaScript data visualization library based on D3 and WebGL that supports a wide range of statistical, scientific, financial, geographic, and 3-dimensional visualizations [Inc15]. The library was initially developed by Plotly Inc. as a core component of their commercial visualization offerings. The library was open sourced under the MIT license in 2015 [Ploc], and may now be used fully offline without requiring any interaction with Plotly Inc's commercial infrastructure.

* Corresponding author: jon.mease@jhuapl.edu

‡ Johns Hopkins Applied Physics Laboratory

Copyright © 2018 Jon Mease. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

```
{
  "data": [
    {
      "type": "bar",
      "y": [2, 3, 1],
      "name": "A",
    },
    {
      "type": "scatter",
      "y": [3, 1, 2],
      "name": "B",
      "marker": {"size": 12}
    }
  ],
  "layout": {"xaxis": {
    "range": [-1, 3],
    "tickvals": [0, 1, 2]
  }}
}
```

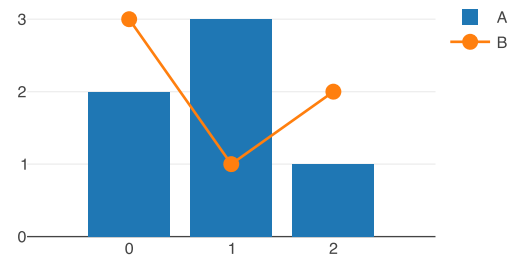


Fig. 1: JSON specification of a basic Plotly.js figure

Data model

Plotly figures are fully defined by a declarative JSON specification. Key components of this specification are shown in the example in Figure 1.

The top-level 'data' property contains an array of the traces present in the figure. The object representing each trace contains a 'type' property that identifies the trace type (e.g. 'scatter', 'bar', 'violin', 'mesh3d', etc.). The remaining properties are used to configure the trace. As of version 1.37.1, Plotly.js supports 32 distinct trace types covering many statistical, scientific, financial, geographic, and 3-dimensional use-cases.

The top-level 'layout' property is an object with properties that specify characteristics of the figure that are independent of its traces. These include the figure's size, axis extents, legend styling, background color, and many others.

Of particular interest to this work, the Plotly.js library is capable of exporting a detailed schema corresponding to this JSON specification. The schema includes the names of all valid

```

{"layoutAttributes":
  ...
  "hovermode": {
    "valType": "enumerated",
    "values": [
      "x",
      "y",
      "closest",
      false
    ],
    "description": "Determines ... "
  }
  ...
}

```

Fig. 2: Plotly.js schema example for the *hovermode* property

properties and information about their permitted values. This schema is the basis for the Plotly rest API [Ploa] and, as discussed below, this schema enables us to use code-generation to generate a complete Python object hierarchy corresponding to the JSON structure. Figure 2 presents a excerpt of the plot schema describing the 'hovermode' property of layout.

Next, we provide a brief overview of the relevant portions of the Plotly.js API that are used by the new widget library. For more information, including detailed method signatures, see [Plob].

Commands

The following Plotly.js commands are used to create and update figures.

`Plotly.newPlot`

Create a new figure with initial traces and layout

`Plotly.restyle`

Update one or more properties of one or more pre-existing traces

`Plotly.layout`

Update one or more properties of the figure's layout

`Plotly.update`

Update both trace and layout properties simultaneously

`Plotly.addTraces`

Add new traces to an existing figure

`Plotly.deleteTraces`

Delete select traces from an existing figure

`Plotly.moveTraces`

Move select traces to a new position in the figure's data array

`Plotly.animate`

Animate property updates in supported trace types

Events

The following events are emitted by Plotly.js figures in response to various kinds of user interaction.

`plotly_restyle`

Emitted when properties of one or more traces are updated. This may either be the result of a `Plotly.restyle` command or the result of user interaction. For example,

clicking on a trace in the legend toggles the trace's visibility in the figure. This visibility state is stored in the top-level `visible` enumeration property on traces.

`plotly_relayout`

Emitted when properties of the figure's layout are updated. This may either be the result of a `Plotly.relayout` command or the result of user interaction. For example, panning or zooming a figure's axis updates the 'range' sub-property of the top-level 'xaxis' and 'yaxis' layout properties.

`plotly_selected`

Emitted when a user completes a selection action using the box select or lasso select tools. The event's data contain the indices of the traces from which points were selected and the indices of the selected points themselves. Similar events are also emitted when a user clicks (`plotly_click`), hovers onto (`plotly_hover`), or hovers off of (`plotly_unhover`) points in a trace.

Variables

The current state of a figure is stored in the following four variables.

`data` and `layout`

These variables store the trace and layout properties explicitly specified by the user.

`_fullData` and `_fullLayout`

These variables store the full collection of trace and layout properties that are currently in use, whether specified by the user or selected by Plotly.js as defaults.

ipywidgets Overview

The ipywidgets library [GFC] provides a useful collection of interactive widgets (sliders, check boxes, radio buttons, etc.) for use in the Jupyter Notebook and in several other contexts [wida]. For the full list of built-in widgets see [widb].

The integration of graphical widgets into the notebook workflow allows users to configure ad-hoc control panels to interactively sweep over parameters using graphical widget controls, rather than by editing code or writing loops over fixed ranges of values.

The infrastructure behind the built-in ipywidgets is available to library authors and many custom ipywidgets libraries have been developed [Cus]. Three notable data visualization examples include bqplot [CSM⁺] for 2-dimensional Grammar of Graphics [Wil05] style visualizations, ipyvolum for 3-dimensional and volumetric visualizations, and ipyleaflet [CG] for geographic visualization.

The high level architecture, shown in Figure 3, consists of four components: The Python model, the JavaScript model, the JavaScript views, and the Comms interface. These components are described below.

Python Model

The Python model is a Python class that inherits from the `ipywidgets.Widget` superclass and uses the `traitlets` library [tra] to declare typed attributes that should be synchronized with the JavaScript model.

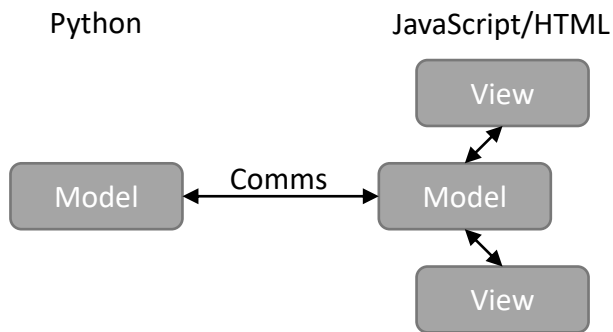


Fig. 3: High level ipywidgets architecture

JavaScript Model

The JavaScript model is a JavaScript class that extends the `@jupyter-widgets/base/WidgetModel` class and declares a collection of attributes that match the traitlet declarations in the corresponding Python model.

When used in the Jupyter Notebook, there is a one-to-one relationship between the Python and JavaScript models. The JavaScript model is constructed just after the Python model is constructed, which may be well before the widget is first displayed.

JavaScript View

The JavaScript view (hereafter referred to as "the view" since there is no ambiguity) is a JavaScript class that extends the `@jupyter-widgets/base/WidgetView` class. When used in the notebook, a separate view is constructed each time a model is displayed. Each view has a reference to one JavaScript model, and multiple views may share the same model.

Comms and Synchronization

The Jupyter Comms API provides an abstraction for performing two-way communication between the front-end and the Python kernel, hiding the complexity of the web server, ZeroMQ, and WebSocket implementation details.

The synchronization of the Python and JavaScript models is accomplished using the widget messaging protocol over the Jupyter Comms infrastructure.

A powerful feature of the widget messaging protocol is that it supports the efficient serialization of nested data structures containing binary buffers. This capability is used by `ipyvolume [Bre]` (and now `plotly.py`) to transfer Python numpy arrays into JavaScript TypedArrays without ASCII encoding.

New Plotly.py Figure API

In `plotly.py` version 3, a figure is represented by an instance of the `plotly.graph_objs.Figure` class. A `Figure` instance maintains an internal representation of the figure's JSON specification, and presents a convenient API for creating and updating this specification.

Code generation is used to create a rich hierarchy of Python classes that correspond to the object hierarchy specified in the plot schema described above. Figure 4 presents an example of property tab completion (a), a property docstring (b), and a validation error message (c) for the `'hovermode'` property of layout that is defined by the schema excerpt in Figure 2.

Select components of the new API are described below, and an example of their use is presented in Figure 5.

Construction

If the full specification of the desired figure is known in advance, the specification may be passed directly to the `Figure` constructor as a Python dict. This construction process will trigger the validation of all properties and nested properties according to the plot schema. Figure 5 (a) presents an example of constructing a `Figure` with a single bar trace.

Property Assignment

A `Figure`'s properties may be configured iteratively after construction using property assignment. Figure 5 (b) presents an example of setting the x-axis range to `[-1, 3]` using property assignment.

Add Traces

A new trace may be added to an existing `Figure` using the `add_{trace}` method that corresponds to the desired trace type. Figure 5 (c) presents an example of adding a new `scatter` trace to a `Figure` instance using the `add_scatter` method.

Batch Update

Multiple properties may be updated simultaneously using a `Figure.batch_update()` context manager. In this case, all property assignments specified inside the `batch_update` context will be executed simultaneously when the context exits. Figure 5 (d) presents an example of assigning four properties across two traces and the layout inside a `batch_update` context.

Reorder Traces

The ordering of traces in the `Figure`'s data list determines the order in which the traces are displayed in the legend, and the colors that are chosen for traces by default. The trace order can be updated by assigning to the `data` property a list that contains a permutation of the figure's current traces. Figure 5 (e) presents an example of swapping the order of the `bar` and `scatter` traces.

Delete Traces

Traces may be deleted by omitting them from the list of traces that is assigned to a `Figure`'s `data` property. Figure 5 (f) presents an example of deleting the `bar` trace by assigning a list that contains only the `scatter` trace.

Batch Animate

Multiple properties may be updated simultaneously using a `Figure.batch_animate()` context manager. When applied to a `Figure` instance this works just like the `batch_update` context manager. However, when applied to a `FigureWidget` instance (described below) the `Plotly.js` library will attempt to smoothly animate the transition to the new property values. Figure 5 (g) presents an example of animating a change in the `Figure`'s x-axis and y-axis range extents.

```
(a) In [1]: import plotly.graph_objs as go
fig = go.FigureWidget(
    data=[go.Bar(y=[2, 3, 1])]

(b) In [ ]: fig.layout.hover|
fig.layout.hoverdistance
fig.layout.hoverlabel
fig.layout.hovermode

(c) In [ ]: fig.layout.hovermode|
Type:          property
String form:   <property object at 0x106cf8ea8>
Docstring:
Determines the mode of hover interactions.

The 'hovermode' property is an enumeration that may be specified as:
- One of the following enumeration values:
  ['x', 'y', 'closest', False]

Returns
-----
Any

(d) In [2]: fig.layout.hovermode = 'nearest'

-----
ValueError                                Traceback (most recent call last)
<ipython-input-2-5d15804abeb3> in <module>()
----> 1 fig.layout.hovermode = 'nearest'
...
ValueError:
Invalid value of type 'builtins.str' received for the 'hovermode' property of layout
Received value: 'nearest'

The 'hovermode' property is an enumeration that may be specified as:
- One of the following enumeration values:
  ['x', 'y', 'closest', False]
```

Fig. 4: Tab completion, documentation, and validation of `hovermode` property

New Plotly.py ipywidgets Implementation

The entry point for the new ipywidgets support is the `plotly.graph_objs.FigureWidget` class. `FigureWidget` is a subclass of `Figure` and, as such, inherits all of the `Figure` characteristics described in the previous section.

Implementing a custom ipywidgets library for Plotly.js presents some architectural challenges. Plotly.js does not expose a model-view separation, each figure stores its own data locally in the figure's root DOM element. This means that each ipywidgets JavaScript view will necessarily be an independent Plotly.js figure instance with its own data. As such, we must take responsibility for keeping the JavaScript model in sync with the state of the Plotly.js figures in each view.

An additional performance-based architectural restriction is that as few properties as possible should be transferred between the Python and JavaScript models. This restriction eliminates solutions that require serialization of the entire plot specification when only a subset of the properties are modified.

The following sections describe our solution to these challenges.

Python to JavaScript Synchronization

Python to JavaScript synchronization is achieved by translating Python `FigureWidget` mutation operations into Plotly.js API commands. These commands, and their associated data, are transferred to the JavaScript model and views using the widget messaging protocol, over the Jupyter Comms infrastructure, as described above. The views are updated by executing the specified Plotly.js command, and the JavaScript model is updated manually in a consistent fashion.

Construction

Construction operations are translated into `Plotly.newPlot` commands. Figure 6 (a) presents an example of the `newPlot` command that results from the construction operation in Figure 5 (a) if the `Figure` class is replaced by `FigureWidget`.

Property Assignment

Trace property assignments are translated into `Plotly.restyle` commands, and layout property assignments are translated into `Plotly.relayout` commands. Figure 6 (b) presents an example of the `relayout` command that results from the property assignment operation in Figure 5 (b).

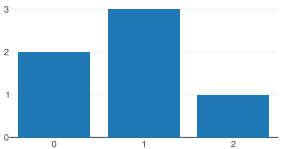
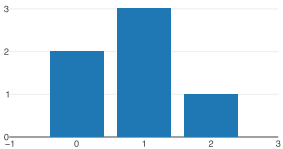
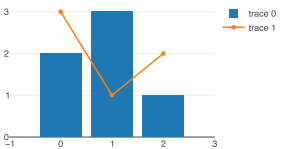
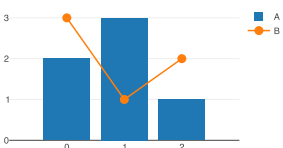
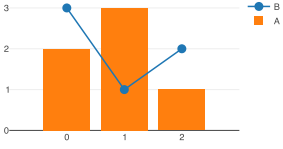
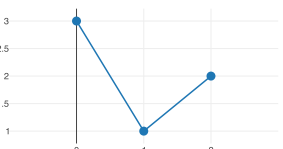
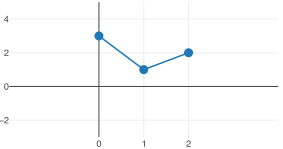
	Code	JSON Specification	Display
(a)	<pre>>>> import plotly.graph_objs as go >>> fig = go.FigureWidget(data=[go.Bar(y=[2, 3, 1])])</pre>	<pre>{ "data": [{ "type": "bar", "y": [2, 3, 1]}, { "layout": {} }</pre>	
(b)	<pre>>>> fig.layout.xaxis.range = [-1, 3]</pre>	<pre>{ "data": [{ "type": "bar", "y": [2, 3, 1]}, { "layout": { "xaxis": { "range": [-1, 3] } } }</pre>	
(c)	<pre>>>> fig.add_scatter(y=[3, 1, 2])</pre>	<pre>{ "data": [{ "type": "bar", "y": [2, 3, 1]}, { "type": "scatter", "y": [3, 1, 2]}, { "layout": { "xaxis": { "range": [-1, 3] } } }</pre>	
(d)	<pre>>>> with fig.batch_update(): ... fig.data[0].name = 'A' ... fig.data[1].name = 'B' ... fig.data[1].marker.size = 12 ... fig.layout.xaxis.tickvals = \ ... [0, 1, 2]</pre>	<pre>{ "data": [{ "type": "bar", "y": [2, 3, 1], "name": "A"}, { "type": "scatter", "y": [3, 1, 2], "name": "B", "marker": { "size": 12 } }], { "layout": { "xaxis": { "range": [-1, 3], "tickvals": [0, 1, 2] } } }</pre>	
(e)	<pre>>>> fig.data = \ ... [fig.data[1], fig.data[0]]</pre>	<pre>{ "data": [{ "type": "scatter", ... }, { "type": "bar", ... }], { "layout": { "xaxis": { "range": [-1, 3], "tickvals": [0, 1, 2] } } }</pre>	
(f)	<pre>>>> fig.data = [fig.data[0]]</pre>	<pre>{ "data": [{ "type": "scatter", ... }, { "layout": { "xaxis": { "range": [-1, 3], "tickvals": [0, 1, 2] } } }</pre>	
(g)	<pre>>>> with fig.batch_animate(): ... fig.layout.xaxis.range = \ ... [-2, 4] ... fig.layout.yaxis.range = \ ... [-3, 5]</pre>	<pre>{ "data": [{ "type": "scatter", ... }, { "layout": { "xaxis": { "range": [-1, 3], "tickvals": [0, 1, 2] }, "yaxis": { "range": [-3, 5] } } }</pre>	

Fig. 5: New Figure API Example

	Plotly.js Command	Arguments
(a)	<code>Plotly.newPlot</code>	<pre>{ "data": [{ "type": "bar", "y": [2, 3, 1] }], "layout": {} }</pre>
(b)	<code>Plotly.relayout</code>	<pre>{ "xaxis.range": [-1, 3] }</pre>
(c)	<code>Plotly.addTraces</code>	<pre>{ "type": "scatter", "y": [3, 1, 2] }</pre>
(d)	<code>Plotly.update</code>	<pre>{ "data": { "name": ["A", "B"], "marker.size": [undefined, 12] }, "layout": { "xaxis.tickvals": [0, 1, 2] } }</pre>
(e)	<code>Plotly.moveTraces</code>	<pre>{ "traceInds": [0, 1], "newTraceIndes": [1, 0] }</pre>
(f)	<code>Plotly.deleteTraces</code>	<pre>{ "traceInds": [1] }</pre>
(g)	<code>Plotly.animate</code>	<pre>{ "layout": { "xaxis.range": [-1, 3], "yaxis.range": [-3, 5] } }</pre>

Fig. 6: Plotly.js commands corresponding to operations in Figure 5 if the `Figure` class is replaced by `FigureWidget`

Add Traces

Add trace operations are translated into `Plotly.addTraces` commands. Figure 6 (c) presents an example of the `addTraces` command that results from the `add_scatter` operation in 5 (c).

Batch Update

Batch update operations are translated in to `Plotly.update` commands. Figure 6 (d) presents an example of the `update` command that results from the `batch_update` operation in 5 (d).

Reorder Traces

Trace reordering operations are translated into `Plotly.moveTraces` commands. Figure 6 (e) presents an example of the `moveTraces` command that results from the data assignment operation in 5 (e).

Delete Traces

Trace deletion operations are translated into `Plotly.deleteTraces` commands. Figure 6 (f) presents an example of the `deleteTraces` command that results from the data assignment operation in 5 (f).

Batch Animate

Batch animate operations are translated into `Plotly.animate` commands. Figure 6 (g) presents an example of the `animate`

command that results from the `batch_animate` operation in 5 (g).

JavaScript to Python Synchronization

JavaScript to Python synchronization is required when a user interacts with a Plotly.js figure in a view in such a way that the figure's internal specification is modified. For example, the action of zooming or panning a figure causes a modification to the figure's x-axis and y-axis range properties.

To maintain consistency, views listen for `plotly_restyle` and `plotly_relayout` events and forward these commands to the Python model. The Python model then applies the command to itself and forwards the command to the Java Script model and any additional views.

Property change callbacks

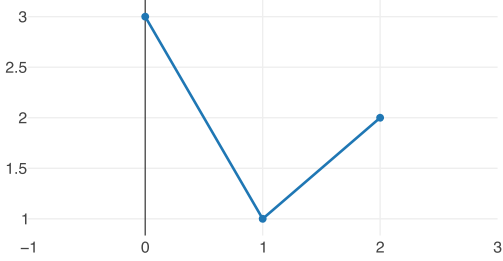
Python functions may be registered for execution when particular trace or layout properties are modified by using the `on_change` method. This method is available on all compound objects in the figure hierarchy.

Figure 7 presents an example of constructing and displaying a `FigureWidget` instance (a) and then registering the `handle_zoom` function for execution when the range sub-property of either the `xaxis` or the `yaxis` properties is changed (b).


```

>>> import plotly.graph_objs as go
>>> from IPython.display import display
>>> fig = go.FigureWidget(
    data=[go.Scatter(y=[3, 1, 2])],
    layout={'xaxis': {'range': [-1, 3]}})
>>> display(fig)

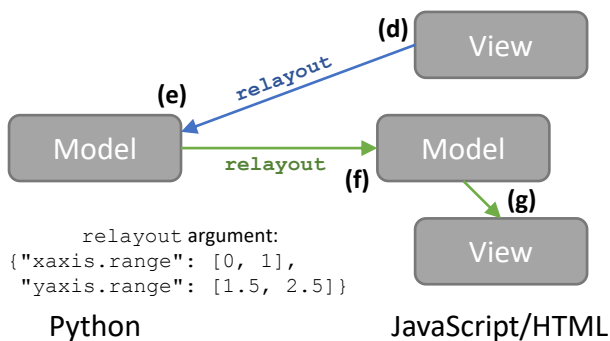
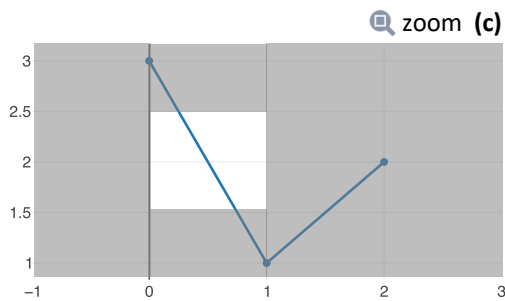
```



```

>>> def handle_zoom(layout, xrange, yrange):
...     print('new x-range:', xrange)
...     print('new y-range:', yrange)
>>> fig.layout.on_change(handle_zoom,
    'xaxis.range',
    'yaxis.range')

```



Python

JavaScript/HTML

```

new x-range: (0, 1)
new y-range: (1.5, 2.5)

```

Fig. 7: Zoom property change callback example

Next, the zoom tool is used to select a region that extends from 0 to 1 on the x-axis and from 1.5 to 2.5 on the y-axis (c). The Plotly.js figure that executes the zoom action emits a `plotly_relayout` event (d) which the view forwards to the Python model (e). The Python model applies the update to itself and then sends a `relayout` message to the JavaScript model (f) and any additional JavaScript views (g). Finally, the Python model executes any callback functions registered on the `range` sub-property of `xaxis` or `yaxis` (h).

Point interaction callbacks

As discussed above, a Plotly.js figure emits events when a user interacts with a trace by clicking (`plotly_click`), hovering onto (`plotly_hover`), hovering off of (`plotly_unhover`), or selecting (`plotly_selected`) points. Trace objects in `plotly.py` now support the registration of Python callbacks to be executed when these events occur.

Figure 8 presents an example of constructing and displaying a `FigureWidget` instance with a `scattergl` trace containing 100,000 normally distributed points (a). The `scattergl` trace is a WebGL optimized version of the SVG-based `scatter` trace used in previous examples.

Trace markers are configured to be colored based on a color scale and a numeric vector. The `cmin` and `cmax` properties specify that `color` values of 0 should be mapped to the bottom of the color scale (light gray for the default scale) and values of 1 should be mapped to the top of the color scale (dark red for the default scale). The color vector is initialized to all zeros so all points are initially light gray in color.

Next, the `brush` function is defined and then registered with the trace for execution when a selection event occurs using the trace's `on_selection` method (b). The first argument to the `brush` function is the trace that was selected (the `scattergl` trace in this case) and the second argument is a list of the indices of the points that were selected.

The box select tool is used to select a rectangular region of points (c). This triggers the execution of the `brush` function. The `brush` function updates the marker's `color` property to be an array where the elements corresponding to selected points have a value of 1 and all other elements have a value of 0. Due to the marker color configuration described above, this causes the selected points to be displayed in dark red.

It is significant to note that even though there are 100,000 points, the time to display the initial figure and the time to update point colors based on a new selection are each less than one second. This latency level is enabled by the efficient transfer of numpy arrays to the JavaScript front-end as binary buffers over the Jupyter Comms interface, and by the WebGL accelerated implementation of the `scattergl` trace.

Default Properties

Plotly.js provides a flexible range of configuration options to control the appearance of a figure's traces and layout, and it will attempt to compute reasonable defaults for properties not specified by the user.

To improve the experience of interactively refining a figure's appearance, it is very helpful to provide the user with the default values of unspecified properties. For example, if a user would like to specify a `scatter` trace marker size that is slightly larger than the default, it is very helpful for the user to know that the default value is 6.

Default property information for traces may be determined by comparing the `data` and `_fullData` variables of the Plotly.js figure. Any property value specified in `_fullData` that is not specified in `data` is considered a default property value. Similarly, the `layout` and `_fullLayout` variables may be used to determine default values for layout properties.

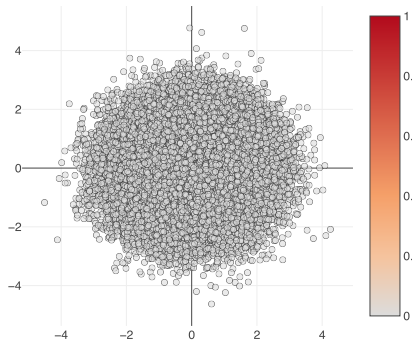
Default properties are transferred from a view to the Python model upon any change to the Plotly.js figure. These default property values are then returned by the Python model during property access when no user specified value is available.

```

>>> import plotly.graph_objs as go
>>> import numpy as np
>>> from IPython.display import display
>>> N = 100000
>>> fig = go.FigureWidget(
...     data = [
...         go.Scattergl(
...             x = np.random.randn(N),
...             y = np.random.randn(N),
...             mode = 'markers',
...             marker={
...                 'color': np.zeros(N),
...                 'opacity': 0.6,
...                 'cmin': 0, 'cmax': 1,
...                 'line': {'width': 1},
...                 'showscale': True}),
...         layout = {'width': 500,
...                    'height': 500})
>>> display(fig)

```

(a)



```

>>> def brush(trace, points, *_):
...     inds = np.array(points.point_inds)
...     selected = np.zeros(N)
...     if inds.size:
...         selected[inds] = 1
...         trace.marker.color = selected

```

```

>>> fig.data[0].on_selection(brush)

```

(b)

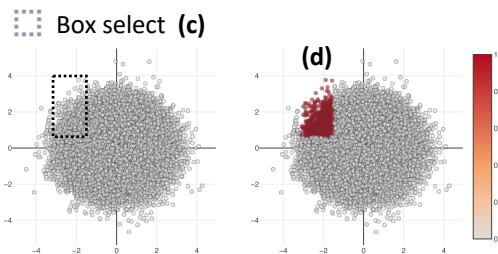


Fig. 8: Data selection and brushing example

Conclusion

The integration of plotly.py version 3 with the ipywidgets library brings a wide range of benefits to plotly.py users working in the Jupyter Notebook. Figure properties are now easily discoverable through the use of tab completion, and they are understandable thanks to the presence of detailed docstrings. This greatly reduces the need for users to interrupt their analysis workflow to consult external documentation resources.

All of these properties may be updated using property assignment syntax and the updates are immediately applied to all

of the displayed views of the figure. This allows users to begin the visualization process with simple figures, and then iteratively refine them.

These iterative updates transfer as few properties from Python to JavaScript as possible, and numpy arrays are transferred as binary buffers without ASCII encoding. Combined with the Plotly.js library's performance optimized WebGL trace types, this allows users to create and interactively explore visualizations of data sets with hundreds of thousands of points.

Plotly figures may now be arranged in custom layouts with other ipywidgets, and Python functions may now be registered for execution in response to figure interactions including pan, zoom, click, hover, and selection. These features allow users to create rich dashboards right in the notebook.

In total, the integration of ipywidgets support in plotly.py version 3 dramatically enhances the interactive data visualization experience for plotly.py users working in the Jupyter Notebook, and we are excited to see what the SciPy community will build with these new tools.

Acknowledgements

The development of the ipywidgets integration was supported by the Johns Hopkins Applied Physics Laboratory. The integration of this work into plotly.py version 3 was additionally supported by Plotly Inc.

REFERENCES

- [Bre] Maarten Breddels. maartenbreddels/ipyvolume: 3d plotting for Python in the Jupyter notebook based on IPython widgets using WebGL. URL: <https://github.com/maartenbreddels/ipyvolume>.
- [CG] Sylvain Corlay and Brian Granger. jupyter-widgets/ipyleaflet: A Jupyter - Leaflet.js bridge. URL: <https://github.com/jupyter-widgets/ipyleaflet>.
- [CSM⁺] Sylvain Corlay, Srinivas Sunkara, Dhruv Madeka, Romain Menegaux, Chakri Cherukuri, and Jason Grout. bloomberg/bqplot: Plotting library for IPython/Jupyter Notebooks. URL: <https://github.com/bloomberg/bqplot>.
- [Cus] Project Jupyter | Widgets. URL: <http://jupyter.org/widgets>.
- [GFC] Jason Grout, Jonathan Frederic, and Sylvain Corlay. ipywidgets: Interactive widgets for the Jupyter Notebook. URL: <https://github.com/jupyter-widgets/ipywidgets>.
- [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, 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.
- [Ploa] Plotly REST API, v2. URL: <https://api.plot.ly/v2/plot-schema>.
- [Plob] Plotly.js Function Reference. URL: <https://plot.ly/javascript/plotlyjs-function-reference/>.
- [Ploc] Plotly.js Open-Source Announcement. URL: <https://plot.ly/javascript/open-source-announcement/>.
- [tra] Traitlets — traitlets 4.3.2 documentation. URL: <https://traitlets.readthedocs.io/en/stable/>.
- [wida] Embedding Jupyter Widgets in Other Contexts than the Notebook — Jupyter Widgets 7.2.1 documentation. URL: <https://ipywidgets.readthedocs.io/en/latest/embedding.html>.
- [widb] Widget List — Jupyter Widgets 7.2.1 documentation. URL: <http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html>.
- [Wil05] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005.

WrightSim: Using PyCUDA to Simulate Multidimensional Spectra

Kyle F Sunden^{‡,*}, Blaise J Thompson[‡], John C Wright[‡]

Abstract—Nonlinear multidimensional spectroscopy (MDS) is a powerful experimental technique used to interrogate complex chemical systems. MDS promises to reveal energetics, dynamics, and coupling features of and between the many quantum-mechanical states that these systems contain. In practice, simulation is typically required to connect measured MDS spectra with these microscopic physical phenomena. We present an open-source Python package, `WrightSim`, designed to simulate MDS. Numerical integration is used to evolve the system as it interacts with several electric fields in the course of a multidimensional experiment. This numerical approach allows `WrightSim` to fully account for finite pulse effects that are commonly ignored. `WrightSim` is made up of modules that can be exchanged to accommodate many different experimental setups. Simulations are defined through a Python interface that is designed to be intuitive for experimentalists and theorists alike. We report several algorithmic improvements that make `WrightSim` faster than previous implementations. We demonstrated the effect of parallelizing the simulation, both with CPU multiprocessing and GPU (CUDA) multithreading. Taken together, algorithmic improvements and parallelization have made `WrightSim` multiple orders of magnitude faster than previous implementations. `WrightSim` represents a large step towards the goal of a fast, accurate, and easy to use general purpose simulation package for multidimensional spectroscopy. To our knowledge, `WrightSim` is the first openly licensed software package for these kinds of simulations. Potential further improvements are discussed.

Index Terms—Simulation, spectroscopy, PyCUDA, numerical integration, Quantum Mechanics, multidimensional

Introduction

Nonlinear multidimensional spectroscopy (MDS) is an increasingly important analytical technique for the analysis of complex chemical material systems. MDS can directly observe fundamental physics that are not possible to record in any other way. With recent advancements in lasers and optics, MDS experiments are becoming routine. Applications of MDS in semiconductor photophysics [CTK⁺15], medicine [FGG⁺09], and other domains [PLMZ18] are currently being developed. Ultimately, MDS may become a key research tool akin to multidimensional nuclear magnetic resonance spectroscopy. [PRK⁺09]

A generic MDS experiment involves exciting a sample with multiple pulses of light and measuring the magnitude of the sample response (the signal). The dependence of this signal on the properties of the excitation pulses (frequency, delay, fluence,

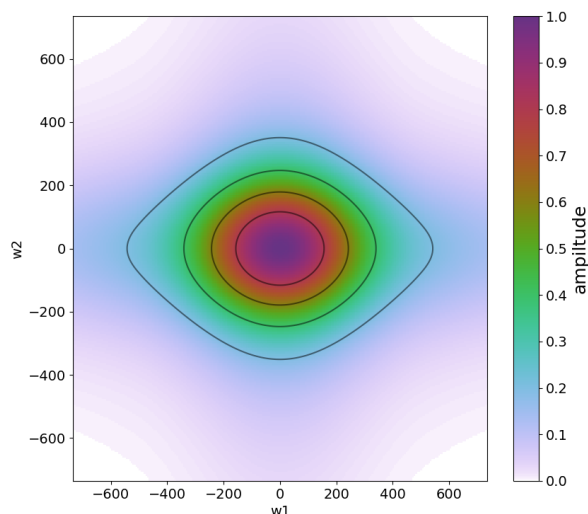


Fig. 1: Simulated spectrum at normalized coordinates

polarization etc.) contains information about the microscopic physics of the material. However, this information cannot be directly "read off" of the spectrum. Instead, MDS practitioners typically compare the measured spectrum with model spectra. A quantitative microscopic model is developed based on this comparison between experiment and theory. Here, we focus on this crucial modeling step. We present a general-purpose simulation package for MDS: `WrightSim`¹.

Figure 1 is a visualization of a spectrum in 2-dimensional frequency-frequency space. The axes are two different frequencies for two separate input electric fields. The system that we have chosen for this simulation is very simple, with a single resonance. The axes are translated such that there is a resonance around 0.0 in both frequencies. This two-dimensional simulation is representative of `WrightSim`'s ability to traverse through many aspects of experimental space. Every conceivable pulse parameter (delay, fluence, frequency, chirp etc.) can become an axis in the

* Corresponding author: sunden@wisc.edu

‡ University of Wisconsin--Madison

Copyright © 2018 Kyle F Sunden et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. Source code available at <https://github.com/wright-group/WrightSim>, released under MIT License.

simulation.

WrightSim is designed with the experimentalist in mind, allowing users to parameterize their simulations in much the same way that they would collect a similar spectrum in the laboratory. WrightSim is modular and flexible. It is capable of simulating different kinds of MDS, and it is easy to extend to new kinds.

WrightSim uses a numerical integration approach that captures the full interaction between material and electric field without making common limiting assumptions. This approach makes WrightSim flexible, accurate, and interpretable. While the numerical approach we use is more accurate, it does demand significantly more computational time. We have focused on performance as a critical component of WrightSim. Here we report algorithmic improvements which have significantly decreased computational time (i.e. wall clock time) relative to prior implementations. We also discuss parallelization approaches we have taken, and show how the symmetry of the simulation can be exploited. While nascent, WrightSim has already shown itself to be a powerful tool, greatly improving execution time over prior implementation.

A Brief Introduction of Relevant Quantum Mechanics

This introduction is intended to very quickly introduce *what* is being done, but not *why*. If you are interested in a more complete description, please refer to Kohler, Thompson, and Wright. [KTW17]

WrightSim uses the density matrix formulation of quantum mechanics. This formulation allows us to describe mixed states (coherences) which are key players in light-matter-interaction and spectroscopy. This involves numerically integrating the Liouville-Neumann equation [Gib02]. This strategy has been described before [GED09], so we are brief in our description here.

WrightSim calculates multidimensional spectra for a given well-defined Hamiltonian. We do not make common limiting assumptions that allow reduction to analytical expressions. Instead, we propagate all of the relevant density matrix elements, including populations and coherences, in a numerical integration. This package does **not** perform *ab initio* computations. This places WrightSim at an intermediate level of theory where the Hamiltonian is known, but accurately computing the corresponding multidimensional spectrum requires complicated numerical analysis.

Now, we focus on one representative experiment and Hamiltonian. In this case, we are simulating the interactions of three electric fields to induce an output electric field. For three fields, there are $3! = 6$ possible time orderings for the pulses to interact and create superpositions or populations in the material system (Figure 2, columns). Within each time ordering, there are several different pathways (Figure 2, rows). In total, there are 16 pathways, represented in Figure 2 as a series of wave mixing energy level (WMEL) diagrams [LA85]. We are restricting this simulation to have two positive interactions (solid up arrows or dashed down arrows) and one negative interaction (dashed up arrow or solid down arrow). Experimentalists isolate this condition spatially using an aperture. They can isolate the time orderings by introducing delays between pulses. Simulation allows us to fully separate each pathway, leading to insight into the nature of pathway interference in the total signal line shape.

Figure 3 shows a finite state automaton for the same system as Figure 2. The nodes are the density matrix elements themselves.

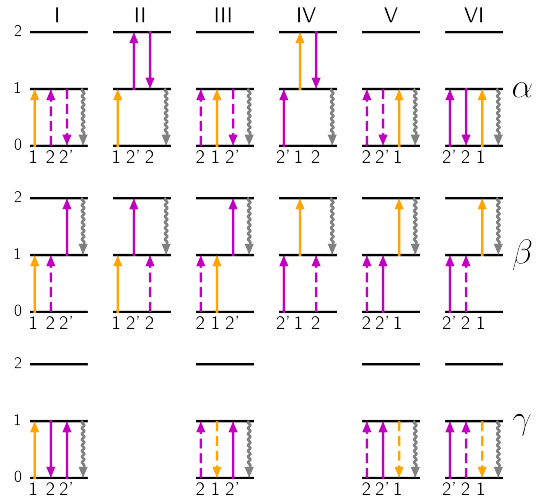


Fig. 2: Independent Liouville pathways simulated. Excitations from ω_1 are in yellow, excitations from $\omega_2 = \omega_{2'}$ are shown in purple. Figure was originally published as Figure 1 of Kohler, Thompson, and Wright [KTW17]

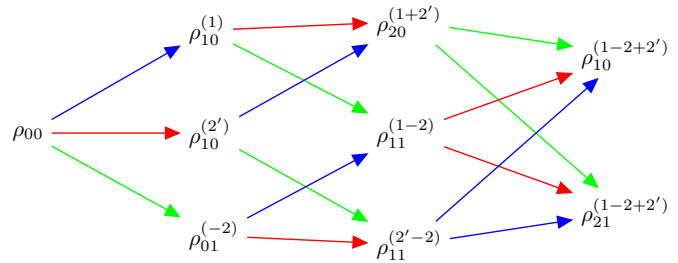


Fig. 3: Finite state automaton of the interactions with the density matrix elements. Matrix elements are denoted by their coherence/population state (the subscript) and the pulses which they have already interacted with (the superscript). Arrows indicate interactions with ω_1 (blue), $\omega_{2'}$ (red), and ω_2 (green). Figure was originally published as Figure S1 of Kohler, Thompson, and Wright [KTW17]

All pathways start at the ground state (ρ_{00}). Encoded within each node is both the quantum mechanical state and the fields with which the system has already interacted. Interactions occur along the arrows, which generate density in the resulting state. Here, the fields must each interact exactly once. Output is generated by the rightmost two nodes, which have interacted with all three fields. These nine states represent all possible states which match the criterion described by the process we are simulating.

We take these nine states and collect them into a state density

vector, $\bar{\rho}$ (Equation 1.1):

$$\bar{\rho} \equiv \begin{bmatrix} \tilde{\rho}_{00} \\ \tilde{\rho}_{01}^{(-2)} \\ \tilde{\rho}_{10}^{(2')} \\ \tilde{\rho}_{10}^{(1)} \\ \tilde{\rho}_{20}^{(1+2')} \\ \tilde{\rho}_{11}^{(1-2)} \\ \tilde{\rho}_{11}^{(2'-2)} \\ \tilde{\rho}_{10}^{(1-2+2')} \\ \tilde{\rho}_{21}^{(1-2+2')} \end{bmatrix}$$

Next we need to describe the transitions within these states. This is the Hamiltonian matrix. Since we have nine states in our density vector, the Hamiltonian is a nine by nine matrix. To simplify representation, six time dependent variables are defined:

$$\begin{aligned} A_1 &\equiv \frac{i}{2} \mu_{10} e^{-i\omega_1 \tau_1} c_1(t - \tau_1) e^{i(\omega_1 - \omega_{10})t} \\ A_2 &\equiv \frac{i}{2} \mu_{10} e^{i\omega_2 \tau_2} c_2(t - \tau_2) e^{-i(\omega_2 - \omega_{10})t} \\ A_{2'} &\equiv \frac{i}{2} \mu_{10} e^{-i\omega_{2'} \tau_{2'}} c_{2'}(t - \tau_{2'}) e^{i(\omega_{2'} - \omega_{10})t} \\ B_1 &\equiv \frac{i}{2} \mu_{21} e^{-i\omega_1 \tau_1} c_1(t - \tau_1) e^{i(\omega_1 - \omega_{21})t} \\ B_2 &\equiv \frac{i}{2} \mu_{21} e^{i\omega_2 \tau_2} c_2(t - \tau_2) e^{-i(\omega_2 - \omega_{21})t} \\ B_{2'} &\equiv \frac{i}{2} \mu_{21} e^{-i\omega_{2'} \tau_{2'}} c_{2'}(t - \tau_{2'}) e^{i(\omega_{2'} - \omega_{21})t} \end{aligned}$$

These variables each consist of a constant factor of $\frac{i}{2}$, a dipole moment term ($\mu_{10|21}$), an electric field phase and amplitude (the first exponential term), an envelope function (c , a Gaussian function here), and a final exponential term which captures the resonance dependence. These variables can then be used to populate the matrix:

$$\bar{Q} \equiv \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -A_2 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_{2'} & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & 0 & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_1 & B_{2'} & -\Gamma_{20} & 0 & 0 & 0 & 0 \\ 0 & A_1 & 0 & -A_2 & 0 & -\Gamma_{11} & 0 & 0 & 0 \\ 0 & A_{2'} & -A_2 & 0 & 0 & 0 & -\Gamma_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 & B_2 & -2A_{2'} & -2A_1 & -\Gamma_{10} & 0 \\ 0 & 0 & 0 & 0 & -A_2 & B_{2'} & B_1 & 0 & -\Gamma_{21} \end{bmatrix}$$

The Γ values along the diagonal represent loss terms such as dephasing (loss of coherence) and population relaxation. To isolate a given time ordering, we can simply set the value of elements which do not correspond to that time ordering to zero.

At each time step, the dot product of the matrix with the $\bar{\rho}$ vector is the change in the $\bar{\rho}$ vector to the next time step (when multiplied by the differential). `WrightSim` uses a second order technique (Runge-Kutta) [BDH06] for determining the change in the $\bar{\rho}$ vector. The core of the simulations is to take the $\bar{\rho}$ vector and multiply by the Hamiltonian at each time step (noting that the Hamiltonian is time dependant, as are the electric fields, themselves). This process repeats over a large number of small time steps, and must be performed separately for any change in the inputs (e.g. frequency [ω] or delay [τ]). As a result, the operation is highly parallelizable. The integration is performed in the rotating frame so the number of time steps can be as small as possible.

Usage

`WrightSim` is designed in a modular, extensible manner in order to be friendly to experimentalists and theorists alike. The key steps to running a basic simulation are:

- Define the experimental space
- Select a Hamiltonian for propagation
- Run the scan
- Process the results

Experimental spaces are defined in an INI format that defines a set of parameters and specifies their defaults and relationships. This can be thought of as a particular experimental setup or instrument.

We use the same experiment and Hamiltonian described above to demonstrate usage. Here, we are using a space called `trive` which provides, among other settings, two independent frequency axes and two independent delay axes, controlling a total of three incident pulses. The frequency axes are called `w1` and `w2`², the delays are `d1` and `d2`. To scan a particular axis, simply set the `points` array to a NumPy [?] array and set its `active` attribute to `True`. You can also set a static value for any available axis, by setting the `points` attribute to a single number (and keeping `active` set to `False`). Finally, the `experiment` class defines the timing of the simulation. Three main parameters control this: `timestep`, which controls the size of each numerical integration step, `early_buffer`, which defines how long to integrate before the first pulse maximum, and `late_buffer`, which defines how long to integrate after the last pulse maximum. Here is an example of setting up a 3D (shape 64x64x32) scan with an additional static parameter set:

```
import WrightSim as ws
import numpy as np

dt = 50. # pulse duration (fs)
nw = 64 # number of frequency points (w1 and w2)
nt = 32 # number of delay points (d2)

# create experiment
exp = ws.experiment.builtin('trive')

# set the scan ranges
exp.w1.points = np.linspace(-500., 500., nw)
exp.w2.points = np.linspace(-500., 500., nw)
exp.d2.points = np.linspace(-2 * dt, 8 * dt, nt)
# tell WrightSim to treat the axis as scanned
exp.w1.active = exp.w2.active = exp.d2.active = True

# set a non-default delay time for the 'd1' axis
exp.d1.points = 4 * dt # fs
exp.d1.active = False

# set time between iterations, buffers
exp.timestep = 2. # fs
exp.early_buffer = 100.0 # fs
exp.late_buffer = 400.0 # fs
```

The Hamiltonian object is responsible for the density vector and holding on to the propagation function used when the experiment is run. Included in the density vector responsibility is the identity of which columns will be returned in the end result array. Hamiltonians may have arbitrary parameters to define themselves in intuitive ways. Under the hood, the Hamiltonian class also holds the C struct and source code for the `PyCUDA` implementation and a method to send itself to the CUDA device. Here is an example

² Note, while the Latin character `w` is used here because it is easier to type in code, it actually represents the Greek letter ω , conventionally, a frequency.

of setting up a Hamiltonian object with restricted pathways and explicitly set recorded element parameters:

```
# create hamiltonian
ham = ws.hamiltonian.Hamiltonian(w_central=0.)

# Select particular pathways
ham.time_orderings = [4, 5, 6]
# Select particular elements to be returned
ham.recorded_elements = [7, 8]
```

Finally, all that is left is to run the experiment itself. The run method takes the Hamiltonian object and a keyword argument `mp`, short for "multiprocess". Any value that evaluates to `False` will run non-multiprocessed (i.e. single threaded). Almost all values that evaluates to `True` with run CPU - multiprocessed with the number of processes determined by the number of cores of the machine. The exception is the special string 'gpu', which will cause `WrightSim` to run using `PyCUDA`.

```
# do scan, using PyCUDA
scan = exp.run(ham, mp='gpu')

# obtain results as a NumPy array
gpuSig = scan.sig.copy()
```

`Running` returns a `Scan` object, which contains several internal features of the scan including the electric field values themselves. The important part, however is the signal array that is generated. In this example, the complex floating point number array is of shape (2x64x64x32) (i.e. the number of `recorded_elements` followed by the shape of the experiment itself). These numbers can be easily manipulated and visualized to produce spectra like that seen in 1. The Wright Group also maintains a library for working with multidimensional data, `WrightTools` [TSM⁺]. This library will be integrated more fully to provide even easier access to visualization and archival storage of simulation results.

Performance

Performance is a critical consideration in the implementation of `WrightSim`. Careful analysis of the algorithms, identifying and measuring the bottlenecks, and working to implement strategies to avoid them are key to achieving the best performance possible. Another key is taking advantage of modern hardware for parallelization. These implementations have their advantages and trade-offs, which are quantified and examined in detail herein.

`NISE` [Gro16] is the package written by Kohler and Thompson while preparing their manuscript [KTW17]. `NISE` uses a slight variation on the technique described above, whereby they place a restriction on the time ordering represented by the matrix, and can thus use a seven element state vector rather than a 9 element state vector. This approach is mathematically equivalent to that presented above. `NISE` is included here as a reference for the performance of previous simulations of this kind.

Algorithmic Improvements

When first translating the code from `NISE` into `WrightSim`, we sought to understand why it took so long to compute. We used Python's standard library package `cProfile` to produce traces of execution, and visualized them with `SnakeViz` [jif17]. Figure 4 shows the trace obtained from a single-threaded run of `NISE` simulating a 32x32x16 frequency-frequency-delay space. This trace provided some interesting insights into how the algorithm could be improved. First, 99.5% of the time is spent inside of a loop which is highly parallelizable. Second, almost one third of

that time was spent in a specific function of NumPy, `ix_`. Further inspection of the code revealed that this function was called in the very inner most loop, but always had the same, small number of parameters. Lastly, approximately one tenth of the time was spent in a particular function called `rotor` (the bright orange box in Figure 4). This function computed $\cos(\theta) + 1j * \sin(\theta)$, which could be replaced by the equivalent, but more efficient $\exp(1j * \theta)$. Additional careful analysis of the code revealed that redundant computations were being performed when generating matrices, which could be stored as variables and reused.

When implementing `WrightSim`, we took into account all of these insights. We simplified the code for matrix generation and propagation by only having the one 9 by 9 element matrix rather than two 7 by 7 matrices. The function that took up almost one third the time (`ix_`) was removed entirely in favor of a simpler scheme for denoting which values to record, simply storing a list of the indices directly. We used variables to store the values needed for matrix generation, rather than recalculating each element. As a result, solely by algorithmic improvements, almost an order of magnitude speedup was obtained (See Figure 5). Still, 99% of the time was spent within a highly parallelizable inner loop.

CPU and GPU Parallel Implementations

`NISE` already had, and `WrightSim` inherited, CPU multiprocessed parallelism using the Python standard library multiprocessing interface. Since almost all of the program is parallelizable, this incurs a four times speedup on a machine with four processing cores (limited more by the operating system scheduling other tasks than by Amdahl's law). This implementation required little adjustment outside of minor API tweaks.

In order to capitalize on the highly parallelizable nature of our multidimensional simulation, the algorithm was re-implemented using Nvidia CUDA [NBGS08]. In order to make the implementation as easy to use as possible, and maintainable over the lifetime of `WrightSim`, `PyCUDA` [KPL⁺12] was used to integrate the call to a CUDA kernel from within Python. `PyCUDA` allows the source code for the device side functions (written in C/C++) to exist as strings within the Python source files. These strings are just-in-time compiled (using `nvcc`) immediately prior to calling the kernel. For the initial work with the CUDA implementation, only one Hamiltonian and one propagation function were written, however it is extensible to additional methods. The just-in-time compilation makes it easy to replace individual functions as needed (a simple form of metaprogramming).

The CUDA implementation is slightly different from the pure Python implementation. It only holds in memory the Hamiltonian matrices for the current and next step, where the Python implementation computes all of the matrices prior to entering the loop. This was done to conserve memory on the GPU. Similarly, the electric fields are computed in the loop, rather than computing all ahead of time. These two optimizations reduce the memory overhead, and allow for easier to write functions, without the help of NumPy to perform automatic broadcasting of shapes.

Scaling Analysis

Scaling analysis, tests of the amount of time taken by each simulation versus the number of points simulated, were conducted for each of the following: `NISE` single threaded, `NISE` Multiprocessed using four cores, `WrightSim` Single threaded, `WrightSim` Multiprocessed using four cores, and `WrightSim` CUDA implementation. A machine with an Intel Core i5-7600

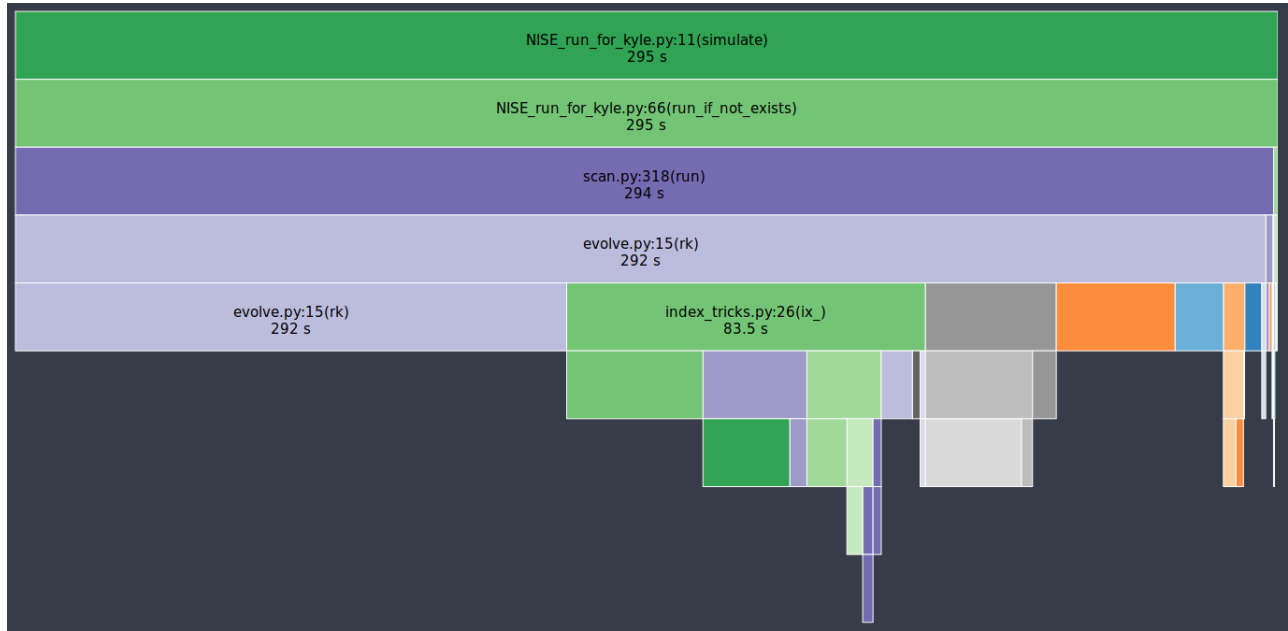


Fig. 4: Profile trace of a single threaded simulation from NISE.

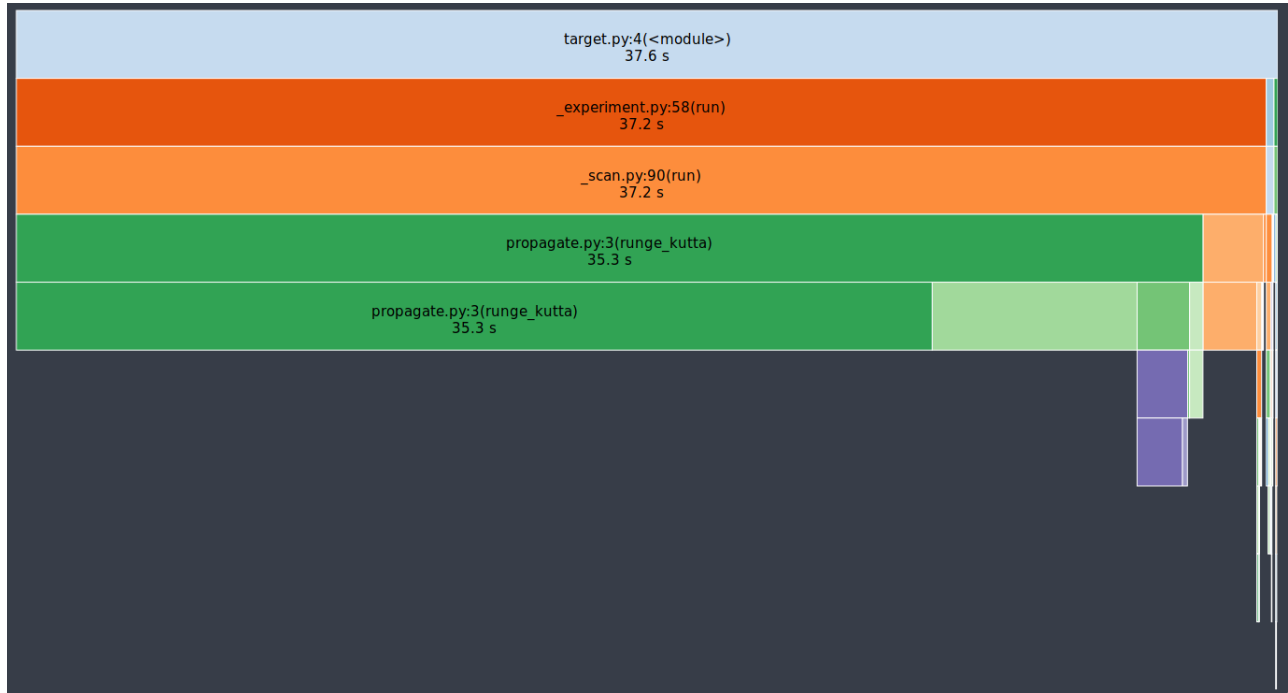


Fig. 5: Profile trace of a single threaded simulation from WrightSim.

(3.5 GHz) CPU and an Nvidia GTX 1060 (3GB) graphics card, running Arch Linux was used for all tests. The simulations were functionally identical, with the same number of time steps and same recorded values. The NISE simulations use two seven by seven matrices for the Hamiltonian, while the WrightSim simulations use a single nine by nine matrix. The results are summarized in Figure 6.

The log-log plot shows that the time scales linearly with number of points. All lines have approximately the same slope at high values of N, though the CUDA implementation grows slower at low N. The Algorithmic improvements alone offer doubled per-

formance over even 4-Core multiprocessed NISE simulation. The CUDA implementation has a positive intercept at approximately 200 milliseconds. This is due, in large part, to the compilation overhead.

Limitations

The CUDA implementation faces limitations at both ends in terms of number of points. On the low side, the cost of compilation and transfer of data makes it slower than the 4-Core CPU Multiprocessing implementation. This crossover point is approximately 256 points (for this simulation, all other parameters being equal). Incidentally, that is also a hard coded block size for the CUDA

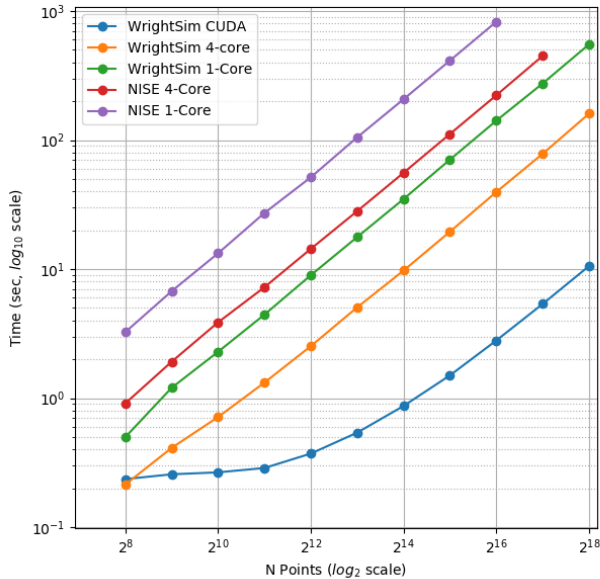


Fig. 6: Scaling Comparison of *WrightSim* and *NISE*

kernel call. While this could be modified to ensure no illegal memory accesses occur on smaller cases, the fact that you are not saving by using CUDA (and even single core performance is under a second) means it is not worth the effort at this time. The hard-coded block size also means that multiples of 256 points must be used in the current implementation.

With larger number of points, we are limited by the amount of memory available to be allocated on the GPU. For each pixel in the simulations presented here, 250 complex numbers represented as doubles must be allocated. Additional space is needed, however it is dominated by this array, which contains the outputs which are then transferred back to the host. Each CUDA thread additionally dynamically allocates the arrays it needs to perform the computation. The current implementation, paired with the particular hardware used, has a limit somewhere between 2^{18} and 2^{19} points. This limit could be increased by using single precision floating point numbers to represent the complex arrays, if the precision trade-off is acceptable (which is yet to be determined).

Future Work

This is still quite early days for *WrightSim*. While it is already a promising proof of concept display of how *PyCUDA* can be applied to this problem, there is still much room for improvement. In general, there are improvements to be made in terms of features, API/ease of use, and indeed further algorithmic improvements.

Features

NISE had implemented a few additional features which were not carried over to *WrightSim* during the initial development efforts which focused on performance thus far.

There was support for chirped electric field pulses, which behave in less ideal fashions than the true sinusoids and Gaussian peaks used thus far. These non-ideal perturbations can have a real effect in spectra collected in the lab, and accurately modelling them helps to interpret these spectra.

Samples in laboratory experiments may have some amount of inhomogeneity within the sample, resulting in broader than would otherwise be expected peaks. This inhomogeneity can be modeled by storing the response array which is calculated by numerical integration, and translating the points slightly. The original *NISE* implementation would perform the simulation multiple times, where that is not needed as a simple translation will do. At one point we considered generating a library of responses in well known coordinates and saving them for future use, avoiding the expensive calculation all together. That seems to be less urgent, given the speed of the CUDA code.

NISE provided a powerful and flexible set of tools to “Measure” the signal, using Fourier transforms and produce arrays that even further mimic what is observed experimentally. That system needs to be added to *WrightSim* for it to be feature-complete. More naïve methods of visualizing work in this case, but a true measurement would allow for richer, more detailed analysis and interpretation.

Some new features could be added, including saving intermediate responses using an HDF5 based file format. The CUDA implementation itself would benefit from some way of saving the compiled code for multiple runs, removing the 0.2 second overhead. Current implementation compiles directly before calling the kernel, whether it has compiled it before or not. If performing many simulations in quick succession (e.g. a simulation larger than the memory allows in a single kernel call) with the same C code, the savings would add up.

The just-in-time compilation enables some special metaprogramming techniques which could be explored. The simple case is using separately programmed functions which have the same signature to do tasks in different ways. Currently there is a small shortcut in the propagation function which uses statically allocated arrays and pointers to those arrays rather than using dynamically allocated arrays. This relies on knowing the size at compilation time. The numbers could be replaced by preprocessor macros which are also fed to the compiler to assign this value “pseudo-dynamically” at compilation time. A much more advanced metaprogramming technique could, theoretically, generate the C struct and Hamiltonian generation function by inspecting the Python code and performing a translation. Such a technique would mean that new Hamiltonians would only have to be implemented once, in Python, and users who do not know C would be able to run CUDA code.

Usability

One of the primary reasons for reimplementing the simulation package is to really think about our interface. As much as possible, the end user should not need to be an experienced programmer to be able to get a simulation. One of the next steps for *WrightSim* is to take a step back and ensure that our API is sensible and easy to follow. We wish to, as much as possible, provide ways of communicating through configuration files, rather than code. Ultimately, a GUI front end may be desirable, especially as the target audience is primarily experimentalists.

Additional Hamiltonians would make the package significantly more valuable as well. To add more Hamiltonians will require ensuring the code is robust, that values are transferred as expected. A few small assumptions were made in the interest of efficiency in the original implementation. Certain values, such as the initial density vector, represented by the Hamiltonian were hard-coded on the device code. While the hard-coded values are reasonable

for most simulations, the ability to set these at run time is desired, and will be added in the future.

Further Algorithmic Improvements

While great strides were taken in improving the algorithms from previous implementations, there are several remaining avenues to gain improved performance in execution time and memory usage. The CUDA implementation is memory bound, both in terms of what can be dispatched, and in terms of time of execution. The use of single precision complex numbers (and other floating point values) would save roughly half of the space. One of the inputs is a large array with parameters for the each electric field at each pixel. This array contains much redundant data, which could be compressed with the parsing done in parallel on the device.

If the computed values could be streamed out of the GPU once computed, while others use the freed space, then there would be almost no limit on the number of points. This relies on the ability to stream data back while computation is still going, which we do not have experience doing, and are not sure CUDA even supports. The values are not needed once they are recorded, so there is no need from the device side to keep the values around until computation is complete.

Additional memory could be conserved by using a bit field instead of an array of chars for determining which time orderings are used as a boolean array. This is relatively minimal, but is a current waste of bits. The Python implementation could potentially see a slight performance bump from using a boolean array rather than doing list searches for this same purpose.

The CUDA implementation does not currently take full advantage of shared cache. Most of the data needed is completely separated, but there are still a few areas where it could be useful.

The current CUDA implementation fills the Hamiltonian with zeros at every time step. The values which are nonzero after the first call are always going to be overwritten anyway, so this wastes time inside of of nested loop. This zeroing could be done only before the first call, removing the nested loop. Additionally, many matrices have a lot of zero values. Often they are triangular matrices, which would allow for a more optimized dot product computation which ignores the zeros in the half which is not populated. Some matrices could even benefit by being represented as sparse matrices, though these are more difficult to use.

Finally, perhaps the biggest, but also most challenging, remaining possible improvement would be to capitalize on the larger symmetries of the system. It's a non-trivial task to know which axes are symmetric, but if it could be done, the amount that actually needs to be simulated would be much smaller. Take the simulation in Figure 1. This was computed as it is displayed, but there are two orthogonal axes of symmetry, which would cut the amount actually needed to replicate the spectrum down by a factor of four. Higher dimensional scans with similar symmetries would benefit even more.

Conclusions

WrightSim, as implemented today, represents the first major step towards a cohesive, easy to use, fast simulation suite for quantum mechanical numerically integrated simulations using density matrix theory. Solely algorithmic improvements enabled the pure Python implementation to be an order of magnitude faster than the previous implementation. The algorithm is highly parallelizable, enabling easy CPU level parallelism. A new implementation provides further improvement than the CPU parallel

code, taking advantage of the General Purpose-GPU Computation CUDA library. This implementation provides approximately 2.5 orders of magnitude improvement over the existing NISE serial implementation. There are still ways that this code can be improved, both in performance and functionality. With WrightSim, we aim to lead by example among the spectroscopic community by providing an open-source package for general-purpose MDS simulation.

REFERENCES

- [BDH06] Paul Blanchard, Robert L Devaney, and Glen R Hall. Numerical Methods. In *Differential Equations*, chapter 7, pages 627–667. Thomson Brooks/Cole, third edition, 2006.
- [CTK⁺15] Kyle J. Czech, Blaise J. Thompson, Schuyler Kain, Qi Ding, Melinda J. Shearer, Robert J. Hamers, Song Jin, and John C. Wright. Measurement of ultrafast excitonic dynamics of few-layer MoS₂ using state-selective coherent multidimensional spectroscopy. *ACS Nano*, 9(12):12146–12157, dec 2015. doi: 10.1021/acsnano.5b05198.
- [FGG⁺09] Frederic Fournier, Rui Guo, Elizabeth M. Gardner, Paul M. Donaldson, Christian Loeffeld, Ian R. Gould, Keith R. Willison, and David R. Klug. Biological and biomedical applications of two-dimensional vibrational spectroscopy: Proteomics, imaging, and structural analysis. *Accounts of Chemical Research*, 42(9):1322–1331, sep 2009. URL: <https://doi.org/10.1021/ar900074p>, doi: 10.1021/ar900074p.
- [GED09] Maxim F. Gelin, Dassia Egorova, and Wolfgang Domcke. Efficient calculation of time- and frequency-resolved four-wave-mixing signals. *Accounts of Chemical Research*, 42(9):1290–1298, sep 2009. URL: <http://dx.doi.org/10.1021/ar900045d>, doi: 10.1021/ar900045d.
- [Gib02] J.W. Gibbs. *Elementary Principles in Statistical Mechanics: Developed with Especial Reference to the Rational Foundations of Thermodynamics*. C. Scribner's sons, 1902. URL: <https://books.google.com/books?id=IGMSAAAAIAAJ>.
- [Gro16] Wright Group. Nise: Numerical integration of the schrödinger equation, 2016. URL: <http://github.com/wright-group/NISE>.
- [jif17] jiffyclub. Snakeviz, 2017. URL: <http://jiffyclub.github.io/snakeviz/>.
- [KPL⁺12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, mar 2012. URL: <http://dx.doi.org/10.1016/j.parco.2011.09.001>, doi: 10.1016/j.parco.2011.09.001.
- [KTW17] Daniel D. Kohler, Blaise J. Thompson, and John C. Wright. Frequency-domain coherent multidimensional spectroscopy when dephasing rivals pulsewidth. *The Journal of Chemical Physics*, 147(8):084202, aug 2017. URL: <https://doi.org/10.1063/1.4986069>, doi: 10.1063/1.4986069.
- [LA85] Duckhwan Lee and Andreas C. Albrecht. A unified view of raman, resonance raman, and fluorescence spectroscopy (and their analogues in two-photon absorption). In R. J. H. Clark and R. E. Hester, editors, *Advances in infrared and Raman Spectroscopy*, chapter 4, pages 179–213. London; New York, 1 edition, 1985.
- [NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40, mar 2008. URL: <https://doi.org/10.1145/1365490.1365500>, doi: 10.1145/1365490.1365500.
- [PLMZ18] Megan K. Petti, Justin P. Lomont, Michał Maj, and Martin T. Zanni. Two-dimensional spectroscopy is being used to address core scientific questions in biology and materials science. *The Journal of Physical Chemistry B*, 122(6):1771–1780, feb 2018. URL: <https://doi.org/10.1021/acs.jpcc.7b11370>, doi: 10.1021/acs.jpcc.7b11370.
- [PRK⁺09] Andrei V. Pakoulev, Mark A. Rickard, Kathryn M. Kornau, Nathan A. Mathew, Lena A. Yurs, Stephen B. Block, and John C. Wright. Mixed frequency-/time-domain coherent multidimensional spectroscopy: Research tool or potential analytical method? *Accounts of Chemical Research*, 42(9):1310–1321, sep 2009. doi: 10.1021/ar900032g.
- [TSM⁺] Blaise J. Thompson, Kyle F. Sunden, Darien J. Morrow, Nathan Andrew Neff-Mallon, Kyle J. Czech, Daniel D. Kohler, Tom Parker, and Rachel Swedin. Wrighttools. doi: 10.5281/zenodo.1198904.

Exploring the Extended Kalman Filter for GPS Positioning Using Simulated User and Satellite Track Data

Mark Wickert^{‡*}, Chiranth Siddappa[‡]



Abstract—This paper describes a Python computational tool for exploring the use of the extended Kalman filter (EKF) for position estimation using the Global Positioning System (GPS) pseudorange measurements. The development was motivated by the need for an example generator in a training class on Kalman filtering, with emphasis on GPS. In operation of the simulation framework both user and satellite trajectories are played through the simulation. The User trajectory is input in local east-north-up (ENU) coordinates and satellites tracks, specified by the C/A code PRN number, are propagated using the Python package SGP4 using two-line element (TLE) data available from [Celestrak].

Index Terms—Global positioning system, Kalman filter, Extended Kalman filter,

Introduction

The Global Positioning System (GPS) allows user position estimation using time difference of arrival (TDOA) measurements from signals received from a constellation of 24 medium earth orbit satellites of space vehicles (SVs). The Kalman filter is a popular optimal *state estimation* algorithm [Simon2006] used by a variety of engineering and science disciplines. In particular the extended Kalman filter (EKF) is able to deal with nonlinearities related to both the measurement equations and state vector process update model. The EKF used in GPS has a linear process model, but a nonlinear measurement model [Brown2012]. This paper describes a Python computational tool for exploring the use of the EKF for GPS position estimation using pseudorange measurements. The development was motivated by the need for an example generator in a training class on Kalman filtering, with emphasis on GPS. What is special about the tool created here is that both *User* and satellite trajectories are custom generated for input to a Kalman filter implemented in a Jupyter notebook. The steps followed are logical and clear. You first enter a desired *User* trajectory/route, then choose appropriate *in-view* GPS satellites, and then using actual GPS satellite orbital mechanics information, create a simulated receiver measurement stream. A 3D plot shows you the satellite tracks in space and the User trajectory on the surface of the earth, over time. The Kalman filter code, also defined in the Jupyter notebook, uses the matrix math commonly found in textbooks, but it is easy to follow as we make use of the PEP

465 @ infix operator for matrix multiplication. As the final step, the data set is played through the Kalman filter in earth-centered earth-fixed (ECEF) coordinates. The User trajectory is input in local east-north-up (ENU) coordinates, and the SVs in view by the User to form the location estimate, are specified by the coarse acquisition (C/A) code pseudo-random noise (PRN) number. The ECEF coordinates of the SVs are then propagated using [SGP4] using the two-line element (TLE) data available from [Celestrak], in time step with the User trajectory. The relationship between ECEF and ENU is explained in Figure 1. For convenience, this computational tool, is housed in a Jupyter notebook. Data set generation and 3D trajectory plotting is provided with the assistance of a single module, [GPS_helper].

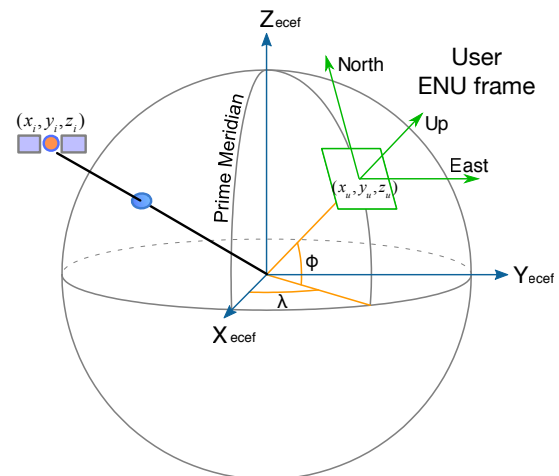


Fig. 1: The earth centric earth fixed (ECEF) coordinate system compared with the local east-north-up (ENU) coordinate system.

GPS Background

GPS was started in 1973 with the first block of satellites launched over the 1978 to 1985 time interval [GPS]. The formal name became NAVSTAR, which stands for NAVigation Satellite Timing And Ranging system, in the early days. At the present time there are 31 GPS satellites in orbit. The original design called for 24 satellites. The satellites orbit at an altitude of about 20,350 km (~12,600 mi). This altitude classifies the satellites as being in a medium earth orbit (MEO), as opposed to low earth orbit

* Corresponding author: mwickert@uccs.edu

‡ University of Colorado Colorado Springs

(LEO), or geostationary above the equator (GEO), or high earth orbit (HEO). The orbit period is 11 hours 58 minutes with six SVs in view at any time from the surface of the earth. Clock accuracy is key to the operation of GPS and the satellite clocks are very accurate. Four satellites are needed for a complete position determination since the user clock is an uncertainty that must be resolved. The maximum SV velocity relative to an earth user is 800m/s (the satellite itself is traveling at ~7000 mph), thus the induced Doppler is up to kHz on the L1 carrier frequency of 1.57542 GHz. This frequency uncertainty plus any motion of the user itself, creates additional challenges in processing the received GPS signals.

Waveform Design and Pseudorange Measurements

Time difference of arrival (TDOA) is the key to forming the User position estimates. This starts by assigning a unique repeating code of 1023 bits to each SV and corresponds to the L1 carrier waveform it transmits. As the User receives the superposition of all the *in-view* satellites, the code known by its PRN number assigned to a particular satellite, is discernable by cross-correlating the composite received L1 signal and a locally generated PRN waveform. The correlation peak and its associated TDOA, become the *pseudorange* or approximate radial distance between the User and SV when multiplied by c , the speed of light.

The pseudorange contains error due to the receiver clock offset from the satellite time and other error components [Brown2012]. The noise-free pseudorange takes the form

$$\rho_i = \sqrt{(x_i - x_u)^2 + (y_i - y_u)^2 + (z_i - z_u)^2} + c\Delta t \quad (1)$$

where (x_i, y_i, z_i) , $i = 1, \dots, 4$, is the satellite ECEF location and (x_u, y_u, z_u) is the user ECEF location, c is the speed of light, and Δt is the receiver offset from satellite time. The product $c\Delta t$ can be thought of as the *range equivalent* timing error. There are three geometry unknowns and time offset, thus at minimum there are four non-linear equations of (1) are what must be solved to obtain the User location.

Solving the Nonlinear Position Equations

Two techniques are widely discussed in the literature and applied in practice [GPS] and [Kaplan]: (1) nonlinear least squares and (2) the extended Kalman filter (EKF). In this paper we focus on the use of the EKF. The EKF is an extension to the linear Kalman filter, so we start by briefly describing the linear model case and move quickly to the nonlinear case.

Kalman Filter and State Estimation

It was back in 1960 that R. E. Kalman introduced his filter [Kalman]. It immediately became popular in guidance, navigation, and control applications. The Kalman filter is an optimal, in the minimum mean-squared error sense, as means to estimate the *state* of a dynamical system [Simon2006]. By state we mean a vector of variables that adequately describes the dynamical behavior of a system over time. For the GPS problem a simplifying assumption regarding the state model is to assume that the User has approximately constant velocity, so a position-velocity (PV) only state model is adequate. The Kalman filter is recursive, meaning that the estimate of the state is refined with each new input measurement and without the need to store all of the past measurements.

Within the Kalman filter we have a *process model* and a *measurement model*. The *process equation* associated with the process model, describes how the state is updated through a state transition matrix plus a process noise vector having covariance matrix \mathbf{Q} . The *measurement model* contains the *measurement equation* that abstractly produces the measurement vector as a matrix times the state vector plus a measurement noise vector having covariance matrix \mathbf{R} . The optimal recursive filter algorithm is formed using the quantities that make up the process and measurement models. For details the reader is referred to the references.

For readers wanting a hands-on beginners introduction to the Kalman filter, a good starting point is the book by Kim [Kim2011]. In Kim's book the Kalman filter is neatly represented input/output block diagram form as shown in Figure 2, with the input being the vector of measurements \mathbf{z}_k , at time k , and the output $\hat{\mathbf{x}}_k$ an updated estimate of the state vector. The Kalman filter variables are defined in Table 1. Note the dimensions seen in Table 1 are n = number of state variables and m = number of measurements.

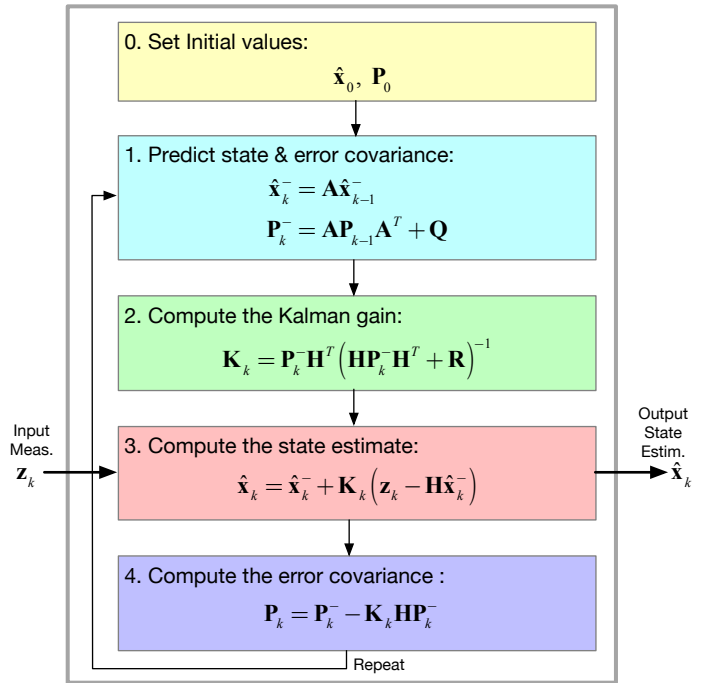


Fig. 2: General Kalman filter block diagram.

State Vector for the GPS Problem

For a PV model the User state vector position and velocity in x, y, z and clock equivalent range and range velocity error [Brown2012]:

$$\begin{aligned} \mathbf{x} &= [x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8] \\ &= [x \ \dot{x} \ y \ \dot{y} \ z \ \dot{z} \ c\Delta t \ c\dot{\Delta}t] \end{aligned} \quad (2)$$

where ECEF coordinates are assumed and the over dots denote the time derivative, e.g., $\dot{x} = dx/dt$. We further assume that there is no coupling between $x, y, z, c\Delta t$, thus the state transition matrix \mathbf{A} is a 4×4 block diagonal matrix of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{cv} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{cv} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{cv} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_{cv} \end{bmatrix} \quad (3)$$

State Estimate (output)	
$\hat{\mathbf{x}}_k$ ($n \times 1$)	State estimate at time k
Measurement (input)	
\mathbf{z}_k ($m \times 1$)	Measurement at time k
System Model	
\mathbf{A} ($n \times n$)	State transition matrix
\mathbf{H} ($m \times n$)	Measurement matrix
\mathbf{Q} ($n \times n$)	State error autocovariance matrix
\mathbf{R} ($m \times m$)	Measurement error autocovariance matrix
Internal Comp. Quant.	
\mathbf{K}_k ($n \times m$)	Kalman gain
\mathbf{P}_k ($n \times n$)	Estimate of error covariance matrix
$\hat{\mathbf{x}}_k^-$ ($n \times 1$)	Prediction of the state estimate
\mathbf{P}_k^- ($n \times n$)	Prediction of error covariance matrix

TABLE 1: The Kalman filter variables and a brief description.

where

$$\mathbf{A}_{cv} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \quad (4)$$

Process Model Covariance Matrix

The process covariance matrix for the GPS problem is a block diagonal Matrix, with three identical blocks for the position-velocity pairs and one matrix for the clock-clock drift pair. The block diagonal form means that the states are assumed be statistically coupled only in pairs and outside of the pairs uncorrelated. In the model of [Brown2012] each position-velocity state-pair has two variance terms and one covariance term describing an upper triangle 2×2 submatrix

$$\mathbf{Q}_{xyz} = \sigma_{xyz}^2 \begin{bmatrix} \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} & \Delta t \end{bmatrix} \quad (5)$$

where σ_{xyz}^2 is a white noise spectral density representing random walk velocity error. The clock state variable pair has a 2×2 covariance matrix governed by S_p , the white noise spectral density leading to random walk velocity error. The clock and clock drift has a more complex 2×2 covariance submatrix, \mathbf{Q}_b , with S_g the white noise spectral density leading to a random walk clock frequency error plus white noise clock drift, thus two components of clock phase error

$$\mathbf{Q}_b = \begin{bmatrix} S_f \Delta t + \frac{S_g \Delta t^3}{3} & \frac{S_g \Delta t^2}{2} \\ \frac{S_g \Delta t^2}{2} & S_g \Delta t \end{bmatrix} \quad (6)$$

In final form \mathbf{Q} is a 4×4 block covariance matrix

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{xyz} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_{xyz} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_{xyz} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{Q}_b \end{bmatrix} \quad (7)$$

Measurement Model Covariance Matrix

The covariance matrix of the pseudorange measurement error is assumed to be diagonal with equal variance σ_r^2 , thus we have

$$\mathbf{R} = \begin{bmatrix} \sigma_r^2 & 0 & 0 & 0 \\ 0 & \sigma_r^2 & 0 & 0 \\ 0 & 0 & \sigma_r^2 & 0 \\ 0 & 0 & 0 & \sigma_r^2 \end{bmatrix} \quad (8)$$

for the case of $m = 4$ measurements. Being diagonal means that all measurements are assumed statistically uncorrelated, which is reasonable.

Extended Kalman Filter

The extended Kalman filter (EKF) allows both the state update equation, Step 1 in Figure 2, to be a nonlinear function of the state, and the measurement model, Step 3 in Figure 2, to be a nonlinear function of the state. Thus the EKF block diagram replaces two expressions in Figure 2 as follows:

$$\mathbf{A}\hat{\mathbf{x}}_{k-1} \longrightarrow \mathbf{f}(\hat{\mathbf{x}}_{k-1}) \quad (9)$$

$$\mathbf{H}\hat{\mathbf{x}}_{k-1}^- \longrightarrow \mathbf{h}(\hat{\mathbf{x}}_{k-1}^-) \quad (10)$$

For the case of the GPS problem we have already seen that the state transition model is linear, thus the first calculation of **Step 1**, predicted state update expression, is the same as that found in the standard linear Kalman filter. For **Step 3**, the state estimate, we need to linearize the equations $\mathbf{h}(\hat{\mathbf{x}}_k^-)$. This is done by forming a matrix of partials or Jacobian matrix, which then generates an equivalent \mathbf{H} matrix as found in the linear Kalman filter, but in the EKF is updated at each iteration of the algorithm.

$$\mathbf{H} = \left. \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \right|_{\mathbf{x}=\hat{\mathbf{x}}_k^-} \quad (11)$$

$$= \begin{bmatrix} \frac{\partial \rho_1}{\partial x} & 0 & \frac{\partial \rho_1}{\partial y} & 0 & \frac{\partial \rho_1}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_2}{\partial x} & 0 & \frac{\partial \rho_2}{\partial y} & 0 & \frac{\partial \rho_2}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_3}{\partial x} & 0 & \frac{\partial \rho_3}{\partial y} & 0 & \frac{\partial \rho_3}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_4}{\partial x} & 0 & \frac{\partial \rho_4}{\partial y} & 0 & \frac{\partial \rho_4}{\partial z} & 0 & 1 & 0 \end{bmatrix} \quad (12)$$

where

$$\frac{\partial \rho_i}{\partial x} = \frac{-(x_i - \hat{x}_1^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{y}_3^-)^2 + (z_i - \hat{z}_5^-)^2}} \quad (13)$$

$$\frac{\partial \rho_i}{\partial y} = \frac{-(y_i - \hat{y}_3^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{y}_3^-)^2 + (z_i - \hat{z}_5^-)^2}} \quad (14)$$

$$\frac{\partial \rho_i}{\partial z} = \frac{-(z_i - \hat{z}_5^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{y}_3^-)^2 + (z_i - \hat{z}_5^-)^2}} \quad (15)$$

for $i = 1, 2, 3$ and 4.

Computational Tool

The Python computational tool is composed of a Jupyter notebook and a helper module `GPS_helper.py`. The key elements of the helper are described in Figure 3. Here we see that the class `GPS_data_source` is responsible for propagating the SVs in view by the User in time-step with a constant velocity *line segment* User trajectory. The end result is a collection of matrices (ndarrays) that contain the ECEF User coordinates as the triples (x_u, y_u, z_u) versus times (also the ENU version) and for each SV indexed as $i = 1, 2, 3, 4$, the ECEF triples (x_i, y_i, z_i) , also as a function of time. The time step value is T_s s.

It is important to note that in creating a data set the developer must choose satellite PRNs that place the SVs in view of the user for the given start time and date. One approach is by trial and error. Pick a particular time and date, choose four PRNs, and produce the data set and create a 3D plot using `GPS_helper.SV_User_Traj_3D()`. This is quite tedious! A better approach is to use a GPS cell phone app, or better yet a

Module: GPS_helper.py	
Class: GPS_data_source	Inputs/Outputs
Constructor():	(0) GPS TLE text file from Celestrak as 'GPS_tle.txt' (1) List of SVs in view by User as 'PRN #' (2) User Reference Location as LAT, LONG, ALT (3) Sampling Period (default = 1s)
user_traj_gen():	(0) Route (a list of 2D nodes in ENU mi) (1) User velocity in mph GMT trajectory start time (2-6): (2) Year (2k year, i.e., 2018 -> 18) (3) Month (4) Day (5) Hour (6) Minute
returns:	(0) User position in ENU (ndarray) vs time (1) User position in ECEF (ndarray) vs time (2) SV position (ndarray) vs time (3) SV velocity (ndarray) vs time
Functions:	Inputs/Outputs
SV_User_Traj_3D(): (displays 3D plot)	(0) GPS data source object (1) SV position ndarray (2) User position ndarray (3) 3D plot ALT = 20 (4) 3D plot AZIM = 20
returns:	none

Fig. 3: Of significance the helper module, GPS_helper.py, contains a class and a 3D plotting function that supports time-varying data set generation of satellite positions and the corresponding User trajectory.

stand-alone GPS that displays a map with PRN numbers of what SVs are in view and their signal strengths. An example from a Garmin GPSmap 60CSx [Garmin] is shown in Figure 4 The time and date used in the simulation then corresponds to the time and date of the actual app measurements. A current TLE set should also be obtained from Celestrak.

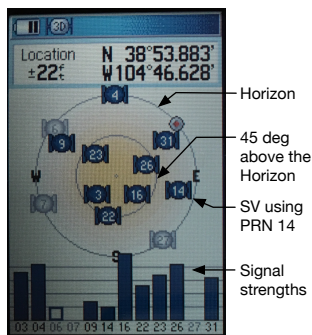


Fig. 4: SV map of satellites in use on a commercial GPS receiver.

With a data set generated the next step is to generate pseudorange measurements, as the real GPS receiver would obtain TDOAs via waveform cross-correlation with a local version of the SVs PRN sequence. Finally, we estimate the user position using the EKF. Classes for both these calculations are contained the Jupyter notebook Kalman_GPS_practice. A brief description of the two classes is given in Figure 5.

The mathematical details of the EKF were discussed earlier, the Python code implementation is found in the public and private methods of the GPS_EKF class. The essence of Figure 2 is the code in the update () method:

```
def next_sample(self, z, SV_Pos):
    """
    Update the Kalman filter state by inputting a
```

Kalman GPS Jupyter Notebook Classes	
Class: GetPseudoRange	Inputs/Outputs
Constructor():	(0) Pseudorange std. dev. (default 0) (1) Pseudo range bias CDt (default 0) (2) Number of satellites in view (default 4)
measurement():	(0) User position in ECEF (ndarray) vs time (1) Satellite (SV) position in ECEF (ndarray) vs time
returns:	none, but USER_SR (ndarray) is filled
Class: GPS_EKF	Inputs/Outputs
Constructor():	(0) User initial position in ECEF (1) Time step (default 1s) (2) Process model diagonal covariance (3) Clock drift random phase walk (default 36) (4) Clock drift random frequency walk (default 0.01) (5) Pseudorange measurement variance (default 36) (6) Number of satellites in view (default 4)
next_sample():	(0) User position ECEF at time step k (1) Satellite (SV) positions ECEF at time step k
returns:	none, none but all EKF attributes updated

Fig. 5: Jupyter notebook classes that synthesize pseudorange test vectors from the time-varying data set created by GPS_helper.py, and implement the extended Kalman filter for estimating the time-varying User position.

```
new set of pseudorange measurements.
Return the state array as a tuple.
Update all other Kalman filter quantities
Input SV ephemeris at one time step, e.g.,
SV_Pos[:, :, i]
"""
# H = Matrix of partials dh/dx
H = self.Hjacob(self.x, SV_Pos)

xp = self.A @ self.x
Pp = self.A @ self.P @ self.A.T + self.Q

self.K = Pp @ H.T @ inv(H @ Pp @ H.T + self.R)

# zp = h(xp), the predicted pseudorange
zp = self.hx(xp, SV_Pos)

self.x = xp + self.K @ (z - zp)
self.P = Pp - self.K @ H @ Pp
# Return the x,y,z position
return self.x[0,0], self.x[2,0], self.x[4,0]
```

Note the above code uses the Python 3.5+ matrix multiplication operator, @, to make the code nearly match the matrix algebra expressions of Figure 2.

Simulation Examples

In this section we consider two examples of using the Python framework to estimate a time-varying User trajectory using a time-varying set of GPS satellites. In the code snippets that follow were extracted from a Jupyter notebook that begins with the magic %pylab inline, hence the namespace is filled with numpy and matplotlib.

We start by creating a line segment user trajectory with ENU tagging, followed by a GPS data source using TLEs date 1/10/2018, and finally, populate User and satellite (SV) ndarrays using the user_traj_gen () method:

```
# Line segment User Trajectory
r11 = [('e', .2), ('n', .4), ('e', -0.1), ('n', -0.2),
      ('e', -0.1), ('n', -0.1)]
```



```

# Create a GPS data source
GPS_ds1 = GPS.GPS_data_source('GPS_tle_1_10_2018.txt',
    Rx_sv_list = \
    ('PRN 32', 'PRN 21', 'PRN 10', 'PRN 18'),
    ref_lla=(38.8454167, -104.7215556, 1903.0),
    Ts = 1)
# Populate User and SV trajectory matrices
# Populate User and SV trajectory matrices
USER_vel = 5 # mph
USER_Pos_enu, USER_Pos_ecf, SV_Pos, SV_Vel = \
    GPS_ds1.user_traj_gen(route_list=r11,
        Vmph=USER_vel,
        yr2=18,
        mon=1,
        day=15,
        hr=8+7, # 1/18/2018
        minute=45) # 8:45 AM MDT

```

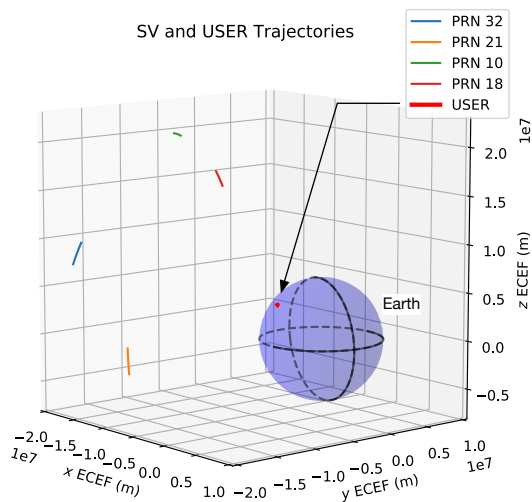


Fig. 6: A 3D plot of the SV trajectories using PRN 32, PRN 21, PRN 10, and PRN 18, and the User trajectory over 13.2 min in ECEF, dated 8:45 AM MDT on 1/18/2018.

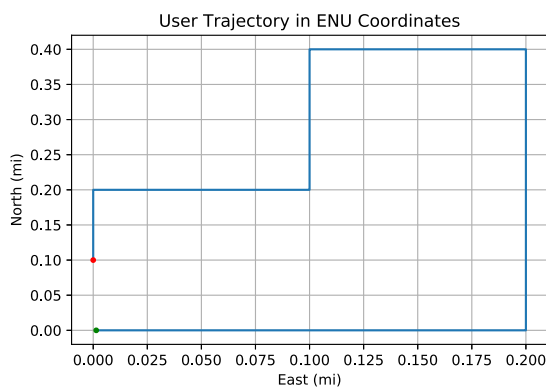


Fig. 7: The ideal user trajectory as defined by `r11` in the above code snippet.

The 3D plot 6 shows clearly the motion of the SVs, even though the simulation run-time is only 13.2 min. The User trajectory on the earth, in this case a location in Colorado Springs, CO appears as a red blob, unless the plot is zoomed in. From the ENU User trajectory we now have a clear view of the route taken by the user. The velocity is only 5 mph in straight line segments.

Case #1

With the data set created we now construct an EKF simulation for estimating the User trajectory from the measured pseudoranges for four SVs. Specifically we consider high quality satellite signals, with measurement update period $T_s = 1$ s, and constant velocity $V_{User} = 5$ mph. The simulation code, as taken from a Jupyter notebook cell, is given below:

```

Nsamples = SV_Pos.shape[2]
print('Sim Seconds = %d' % Nsamples)
dt = 1
# Save user position history
Pos_KF = zeros((Nsamples,3))
# Save history of error covariance matrix diagonal
P_diag = zeros((Nsamples,8))

Pseudo_ranges1 = GetPseudoRange(PR_std=0.1,
    Cdt=0,
    N_SV=4)
GPS_EKF1 = GPS_EKF(USER_xyz_init=USER_Pos_ecf[0,:],
    + 5*randn(3),
    dt=1,
    sigma_xyz=5,
    Sf=36,
    Sg=0.01,
    Rhoerror=36,
    N_SV=4)
for k in range(Nsamples):
    Pseudo_ranges1.measurement(USER_Pos_ecf[k,:],
        SV_Pos[:, :, k])
    GPS_EKF1.next_sample(Pseudo_ranges1.USER_PR,
        SV_Pos[:, :, k])
    Pos_KF[k,:] = GPS_EKF1.x[0:6:2,0]
    P_diag[k,:] = GPS_EKF1.P.diagonal()

```

With the simulation complete, we now consider the ECEF errors in m in Figure 8 for m for (x, y, z) components. The initial position *guess* in this example has a standard deviation of 5 m (or variance of 25 meters-squared), so we see that from the start of the tracking the errors are relatively rather small and then settle down to peak errors of *pm1* m, or so.

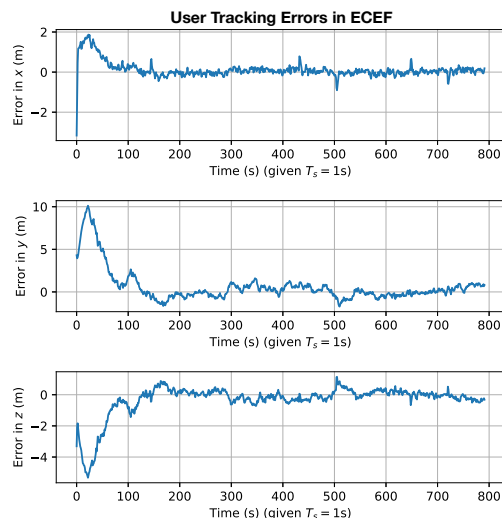


Fig. 8: ECEF errors in position estimation for Case #1.

Figure 9 shows selected error covariance matrix terms from \mathbf{P}_k throughout the simulation. The terms displayed are the position diagonal terms, that is σ_x^2 , σ_y^2 , and σ_z^2 . The initial conditions of the EKF make these variance terms initially large. Settling begins about 50s into the simulation, and the decay continues as the 13.2 m simulation comes to an end. The EKF is behaving as expected.

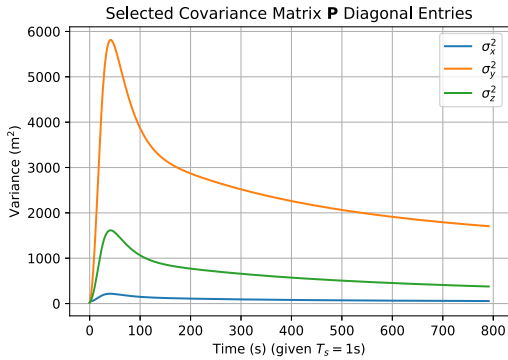


Fig. 9: Selected error covariance matrix terms, in particular the diagonal elements σ_x^2 , σ_y^2 , σ_z^2 .

Finally, in Figure 10 we have a plot of the User trajectory estimate in ENU, as a map-like 2D plot showing just the east-west and north-south axes. The units are tenths of miles, so with the User moving along linear line segments at just 5 mph, the trajectory looks perfect.

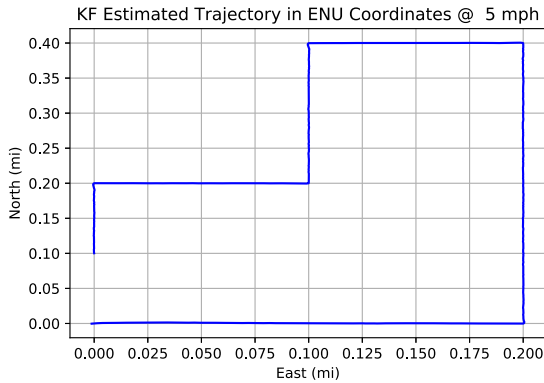


Fig. 10: The estimated user trajectory in ENU coordinates and the same scale as Figure 7.

In the next example parameters will be varied to see the impact.

Case #2

In this case we still consider high quality satellite signals and a 1s update period, but now the user velocity is increased to 30 mph, so the time to traverse the User trajectory is reduced from 13.2 min down to 2.2 min. The random initial (xyz) position is set to a error standard deviation of 50 m compared with 5 m in the first case. We expect to see some difference in performance.

In Figure 11 we again plot the ECEF errors in m. The large initial position error variance forces the plot axes scale to change from Case #1. The initial errors are now very large, but do settle to small values with the exception of blips that occur every time the user changes direction by making a 90° turn. The blips are somewhat artificial, since making a perfect right-angle turn without slowing or *rounding* the corner is more practical. Still it is interesting to see this behavior and also see that the EKF recovers from these errors.

Figure 12 again shows the error covariance terms for σ_x^2 , σ_y^2 , and σ_z^2 . The results here are very similar to Case #1. The variance peaks at about 50 s into the simulation and then rapidly decays. This is not too surprising as the EKF tuning has changed from

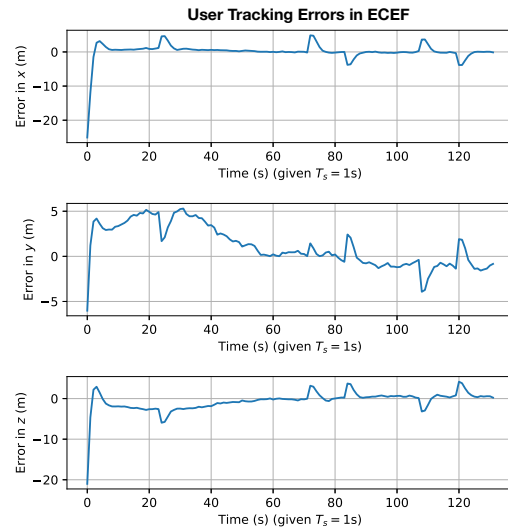


Fig. 11: ECEF errors in position estimation for Case #1.

Case #1, with the exception of the initial position error. Since the simulation only runs for 2.2 min which is 132 s, we have to compare the variances at this time to the Case #2 end results. They appear to be about the same, once again the EKF appears to be working correctly.

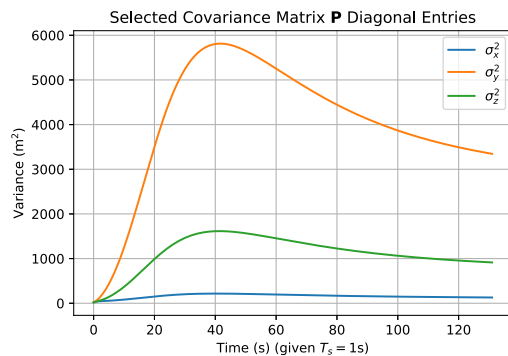


Fig. 12: Selected error covariance matrix terms, in particular the diagonal elements σ_x^2 , σ_y^2 , σ_z^2 .

Finally, Figure 13 plots the ENU trajectory estimate in the plane EN (ignoring the UP coordinate as before). The speed is upped by a factor six compared to case #1. The most notable change is trajectory overshoot at each of the right-angle turns. No surprise here as the EKF is asked to handle very abrupt (and impractical) position changes. The EKF recovers quickly.

Overall the results for both cases are very good. There a lot of *knobs* to turn in this framework, so many options to explore.

It is worthy of note at this point that the *Unscented Kalman Filter* (UKF) [Wan2006], and the more general class of algorithms known as *Sigma-Point Kalman Filters* (SPKF), are today much preferred to the EKF of the past. The EKF is sub-optimal, and the linearization approach makes it sensitive to initial conditions. The EKF requires the Jacobian matrix, which may be hard to obtain, and may not converge without carefully chosen initial conditions. In this paper the EKF was chosen for use in a training scenario because it is the next logical step from the linear Kalman filter, and its development is simple to follow. The UKF is harder to get explain. In the end, the UKF is of similar complexity to the EKF,

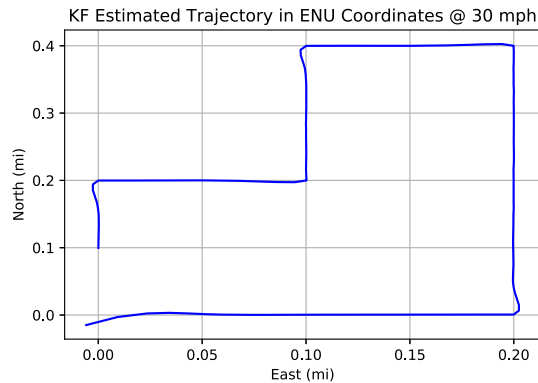


Fig. 13: The estimated user trajectory in ENU coordinates and the same scale as Figure 7.

can offer large performance benefits, and does not require the use of a Jacobian.

Conclusions and Future Work

The objective of creating a Jupyter notebook-based simulation tool for studying the use of the EKF in GPS position estimation has been met. There are many tuning options to explore, which provides a very nice environment for studying a large variety scenarios. The performance results are consistent with expectations.

There are several improvements under consideration. The first is to develop a more realistic user trajectory generator. The second is to make measurement quality a function of the SV range, which would also make the measurement quality SV specific, rather than identical as it is now. A third desire is to move to the UKF to avoid the use of the Jacobian, reduce the sensitivity to initial conditions, and improve performance.

REFERENCES

- [Celestrak] *Celestrak*, (2017, January 26). Retrieved June 26, 2018, from <https://celestrak.com>.
- [SGP4] *Python implementation of most recent SGP4 satellite tracking*, (2018, May 24). Retrieved June 26, 2018, from <https://github.com/brandon-rhodes/python-sgp4>.
- [GPS_helper] *Tools and Examples for GPS*, (2018, June 24), Retrieved from https://github.com/chiranthiddappa/gps_helper.
- [GPS] *Global Positioning System*, (2018, June 24). Retrieved June 26, 2018, from https://en.wikipedia.org/wiki/Global_Positioning_System.
- [Garmin] *GPSMAP® 60CSx with sensors and maps owner's manual*, (2007), Retrieved June 26, 2018, from https://static.garmincdn.com/pumac/GPSMAP60CSx_OwnersManual.pdf.
- [Kalman] Kalman, R. (1960). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 35–45.
- [Brown2012] Brown, R. and Hwang, P. (2012). *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises*, 4th edition. New York: Wiley.
- [Kaplan] Kaplan, E. and Hegarty, C., editors (2017). *Understanding GPS/GNSS: Principles and Applications*, third edition. Boston: Artech House.
- [Kim2011] Phil Kim, P. (2011). *Kalman Filtering for Beginners with MATLAB Examples*. CreateSpace Independent Publishing Platform.
- [Simon2006] Simon, D. (2006). *Optimal State Estimation*. New York: Wiley-Interscience.
- [Wan2006] Wan, E. (2006). Sigma-Point Filters: An Overview with Applications to Integrated Navigation and Vision Assisted Control. *IEEE Nonlinear Statistical Signal Processing Workshop*. doi:10.1109/NSSPW.2006.4378854.

Real-Time Digital Signal Processing Using `pyaudio_helper` and the `ipywidgets`

Mark Wickert^{‡*}

Abstract—The focus of this paper is on teaching real-time digital signal processing to electrical and computer engineers using the Jupyter notebook and the code module `pyaudio_helper`, which is a component of the package `scikit-dsp-comm`. Specifically, we show how easy it is to design, prototype, and test using PC-based instrumentation, real-time DSP algorithms for processing analog signal inputs and returning analog signal outputs, all within the Jupyter notebook. A key feature is that real-time algorithm prototyping is simplified by configuring a few attributes of a `DSP_io_stream` object from the `pyaudio_helper` module, leaving the developer to focus on the real-time DSP code contained in a `callback` function, using a template notebook cell. Real-time control of running code is provided by `ipywidgets`. The PC-based instrumentation aspect allows measurement of the analog input/output (I/O) to be captured, stored in text files, and then read back into the notebook to compare with the original design expectations via `matplotlib` plots. In a typical application `slider` widgets are used to change variables in the callback. One and two channel audio applications as well as algorithms for complex signal (in-phase/quadrature) waveforms, as found in software-defined radio, can also be developed. The analog I/O devices that can be interfaced are both internal and via USB external sound interfaces. The sampling rate, and hence the bandwidth of the signal that can be processed, is limited by the operating system audio subsystem capabilities, but is at least 48 KHz and often 96 KHz.

Index Terms—digital signal processing, `pyaudio`, real-time, `scikit-dsp-comm`

Introduction

As the power of personal computer has increased, the dream of rapid prototyping of real-time signal processing, without the need to use dedicated DSP-microprocessors or digital signal processing (DSP) enhanced microcontrollers, such as the ARM Cortex-M4 [`cortexM4`], can be set aside. Students can focus on the powerful capability of `numpy`, `scipy`, and `matplotlib`, along with packages such as `scipy.signal` [`Scipysignal`] and `scikit-dsp-comm` [`DSPComm`], to explore real-time signals and systems computing.

The focus of this paper is on teaching real-time DSP to electrical and computer engineers using the Jupyter notebook and the code module `pyaudio_helper`, which is a component of the package `scikit-dsp-comm`. To be clear, `pyaudio_helper` is built upon the well known package [`pyaudio`], which has its roots in *Port Audio* [`portaudio`]. Specifically, we show how easy it is to design, prototype, and test using PC-based instrumentation,

real-time DSP algorithms for processing analog signal inputs and returning analog signal outputs, all within the Jupyter notebook. Real-time algorithm prototyping is simplified by configuring a `DSP_io_stream` object from the `pyaudio_helper` module, allowing the developer to quickly focus on writing a DSP `callback` function using a template notebook cell. The developer is free to take advantage of `scipy.signal` filter functions, write custom classes, and as needed utilize global variables to allow the algorithm to maintain *state* between callbacks pushed by the underlying PyAudio framework. The PC-based instrumentation aspect allows measurement of the analog input/output (I/O) to be captured, stored in text files, and then read back into the notebook to compare with the original design expectations via `matplotlib` plots. Real-time control of running code is provided by `ipywidgets`. In a typical application `slider` widgets are used to change variables in the callback during I/O streaming. The analog I/O devices that can be interfaced are both internal and via USB external sound interfaces. The sampling rate, and hence the bandwidth of the signal that can be processed, is limited by the operating system audio subsystem capabilities, but is at least 48 KHz and often 96 KHz.

We will ultimately see that to set up an audio stream requires: (1) create and instance of the `DSP_io_stream` class by assigning valid input and output device ports to it, (2) define a callback function to process the input signal sample frames into output sample frames with a user defined algorithm, and (3) call the method `interactive_stream()` to start streaming.

Analog Input/Output Using DSP Algorithms

A classic text to learn the theory of digital signal processing is [`Opp2010`]. This book is heavy on the underlying theoretical concepts of DSP, including the mathematical modeling of analog I/O systems as shown in Figure 1. This block diagram is a mathematical abstraction of what will be implemented using [`pyaudio`] and a PC audio subsystem. An analog or continuous-time signal $x(t)$ enters the system on the left and is converted to the discrete-time signal $x[n]$ by the analog to digital block. In practice this block is known as the analog-to-digital converter (ADC). The sampling rate f_s , which is the inverse of the sampling period, T , leads to $x[n] = x(nT)$. To be clear, $x[n]$, denotes a sequence of samples corresponding to the original analog input $x(t)$. The use of brackets versus parentheses differentiates the two signal types as discrete-time and continuous-time respectively. The sampling theorem [`Opp2010`] tells us that the sampling rate f_s must be greater than twice the highest frequency we wish to represent

* Corresponding author: mwickert@uccs.edu

‡ University of Colorado Colorado Springs

in the discrete-time domain. Violating this condition results in *aliasing*, which means a signal centered on frequency $f_0 > f_s/2$ will land inside the band of frequencies $[0, f_s/2]$. Fortunately, most audio ADCs limit the signal bandwidth of $x(t)$ in such a way that signals with frequency content greater than $f_s/2$ are eliminated from passing through the ADC. Also note in practice, $x[n]$ is a scaled and finite precision version of $x(t)$. In real-time DSP environments the ADC maps the analog signal samples to signed integers, most likely `int16`. As we shall see in `pyaudio`, this is indeed the case.

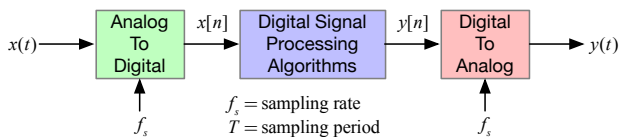


Fig. 1: Analog signal processing implemented using real-time DSP.

The DSP algorithms block can be any operation on samples $x[n]$ that makes sense. Ultimately, once we discuss frame-based processing in the next section, we will see how Python code fulfills this. At this beginning stage, the notion is that the samples flow through the algorithm one at a time, that is one input results in one output sample. The output samples are converted back to analog signal $y(t)$ by placing the samples into a digital-to-analog converter (DAC). The DAC does not simply set $y(nT) = y[n]$, as a continuous function time t must be output. A *reconstruction* operation takes place inside the DAC which *interpolates* the $y[n]$ signal samples over continuous time. In most DACs this is accomplished with a combination of digital and analog filters, the details of which is outside the scope of this paper. The use of

In a DSP theory class the algorithm for producing $y[n]$ from $x[n]$ is typically a *causal* linear time-invariant (LTI) system/filter, implemented via a difference equation, i.e.,

$$y[n] = - \sum_{k=1}^N a_k y[n-k] + \sum_{m=0}^M b_m x[n-m] \quad (1)$$

where $a_k, k = 1, 2, \dots, N$ and $b_m, m = 0, 1, \dots, M$ are the filter coefficients. The filter coefficients that implement a particular filter design can be obtained using design tools in `[DSPComm]`.

Other algorithms of course are possible. We might have a two channel system and perform operations on both signals, say combining them, filtering, and locally generating time varying periodic signals to create audio special effects. When first learning about real-time DSP it is important to start with simple algorithm configurations, so that external measurements can be used to characterize the systems and verify that the intended results are realized. Developing a real-time DSP project follows along the lines of, design, implement, and test using external test equipment. The Jupyter notebook allows all of this to happen in one place, particularly if the test instrumentation is also PC-based, since PC-based instrument results can be exported as `csv` and then imported in Jupyter notebook using `loadtxt`. Here we advocate the use of PC-based instruments, so that all parties, student/instructor/tinkerer, can explore real-time DSP from most anywhere at any time. In this paper we use the Analog Discovery 2 `[AD2]` for signal generation (two function generator channels), signal measurement (two scope channels, with fast Fourier transform (FFT) spectrum analysis included). It is also helpful to have a signal generator cell phone app available, and of course music from a cell phone or PC. All of the cabling is done using 3.5mm

stereo patch cables and small pin header adapters `[3p5mm]` to interface to the AD2.

Frame-based Real-Time DSP Using the `DSP_io_stream` class

The block diagram of Figure 2 illustrates the essence of this paper. Implementing the structure of this figure relies upon the class `DSP_io_stream`, which is housed in `sk_dsp_comm.pyaudio_helper.py`. To make use of this class requires the `scipy` stack (`numpy`, `scipy`, and `matplotlib`), as well as `[DSPComm]` and `[pyaudio]`. `PyAudio` is multi-platform, with the configuration platform dependent. The set-up is documented at `[pyaudio]` and `SPCommTutorial`. The classes and functions of `pyaudio_helper` are detailed in Figure 3. We will make reference to the classes, methods, and functions throughout the remainder of this paper.

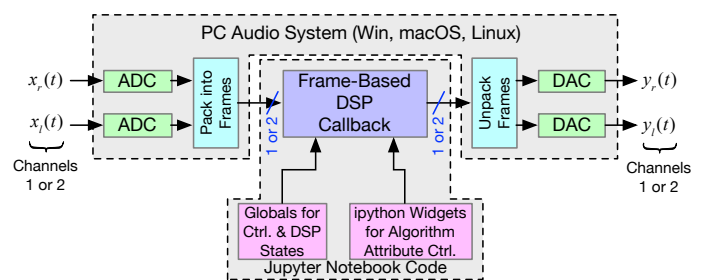


Fig. 2: Two channel analog signal processing implemented using frame-based real-time DSP.

With `DSP_io_stream` one or two channel streaming is possible, as shown in Figure 2. The ADCs and DACs can be internal to the PC or external, say using a USB interface. In a modern PC the audio subsystem has a microphone hardwired to the ADCs and the DACs are connected to the speakers and 3.5mm headphone jack. To provide more flexibility in doing real-time DSP, an external USB audio interface is essential. Two worthy options are the `Sabrent` at less than \$10 and the `iMic` at under \$40. You get what you pay for. The `iMic` is ideal for full two channel audio I/O processing and also has a line-in/mic switch setting, while the `Sabrent` offers a single channel input and two channel output. Both are very capable for their intended purposes. A photograph of the AD2 with the `iMic` interface, 3.5mm splitters and the pin header interfaces mentioned earlier, is shown in Figure 4. The 3.5mm audio splitters are optional, but allow headphones to be plugged into the output while leaving the AD2 scope connected, and the ability to input music/function generator from a cellphone while leaving the AD2 input cable connected (pins wires may need to be pulled off the AD2 to avoid interaction between the two devices in parallel).

When a `DSP_io_stream` is created (top of Figure 3) it needs to know which input and output devices to connect to. If you just want and input or just an out, you still need to supply a valid output or input device, respectively. To list the internal/external devices available on a given PC we use the function `available_devices()` from Figure 3. If you add or remove devices while the notebook kernel is running, you will need to restart the kernel to get an accurate listing of devices. The code block below was run with the `iMic` plugged into a USB hub:

Module: sk_dsp_comm.pyaudio_helper.py	
Class: DSP_io_stream	Inputs/Outputs
Constructor():	(0) Stream callback function name (1) Input device index (default 1) (2) Output device index (default 4) (3) Frame length (default 1024) (4) Sampling rate in Hz (default 44100) (5) Capture buffer length in s (default 0) (6) Sleep time (default 0.1 s from PyAudio)
interactive_stream(): (threaded & buttons)	(0) Stream time in s (default 2, 0 for infinite) (1) Number of channels (default 1 or 2)
returns:	none, but ipywidget start/stop buttons
DSP_callback_tic():	None, but updates a time stamp attribute
returns:	none
DSP_callback_toc():	None, but updates a time stamp attribute
returns:	none
stream_stats():	None
returns:	Prints callback statistics
DSP_capture_add_samples():	(0) Append a new frame of float signal samples to the attribute data_capture
returns:	none
cb_active_plot():	(0) Start time in ms (1) Stop time in ms (2) Line color (default 'b')
returns:	Timing plot showing time in callback
DSP_capture_add_samples_stereo():	(0) Append a new frame of left float signal samples to the attribute data_capture_left (1) Append a new frame of right float signal samples to the attribute data_capture_right
returns:	none
get_LR():	(0) Packed float32 input frame
returns:	(0) Unpacked float32 left channel (1) Unpacked float32 right channel
pack_LR():	(0) Left output float32 frame (1) Right output float32 frame
returns:	(0) Packed float32 frame
Class: loop_audio	Inputs/Outputs
Constructor():	(0) Audio sample array to be looped (1) Offset into array (default 0)
get_samples():	(0) frame_length
Functions:	Inputs/Outputs
available_devices():	None
returns:	Prints available input and output audio devices along with their port indices

Fig. 3: The major classes and functions of the module sk_dsp_comm.pyaudio_helper.py.

```
import sk_dsp_comm.pyaudio_helper as pah
In[3]: pah.available_devices()
Out[3]:
Index 0 device name = Built-in Microphone,
        inputs = 2, outputs = 0
Index 1 device name = Built-in Output,
        inputs = 0, outputs = 2
Index 2 device name = iMic USB audio system,
        inputs = 2, outputs = 2
```

The output list can be viewed as a look-up table (LUT) for how to patch physical devices into the block diagram of Figure 2.

We now shift the focus to the interior of Figure 2 to discuss frame-based DSP and the *Frame-Based DSP Callback*. When a DSP microcontroller is configured for real-time DSP, it can focus on just this one task very well. Sample-by-sample processing is possible with low I/O latency and overall reasonable audio sample

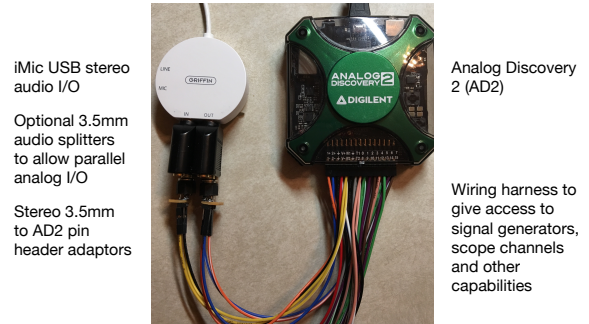


Fig. 4: The iMic stereo USB audio device and the Digilent Analog Discovery 2 (AD2), including the wiring harness.

throughput. On a PC, with its multitasking OS, there is a lot going on. To get reasonable audio sample throughput the PC audio subsystem fills or *packs* an input buffer with `frame_length` samples (or two times `frame_length`, sample for a two channel stream) originating as 16-bit signed integers (i.e., `int16`), before calling the *callback* function. The details of the callback function is the subject of the next section. As the callback prepares to exit, an output buffer of 16-bit signed integers is formed, again of length `frame_length`, and the buffer is absorbed by the PC audio subsystem. In the context of *embedded systems* programming, the callback can be thought of as an *interrupt service routine*. To the PC audio community the frame or buffer just described, is also known as a *CHUNK*. In a two-channel stream the frame holds an interleaving of left and right channels, `...LRLRL...` in the buffer formed/absorbed by the PC audio system. Understand that the efficiency of frame-based processing comes with a price. The buffering either side of the callback block of Figure 2 introduces a latency or processing time delay of at least two times the `frame_length` times the sampling period.

Moving along with this top level discussion, the central block of Figure 2 is labeled *Frame-Based DSP Callback*, and as we have alluded to already, is where the real-time DSP code resides. Global variables are needed inside the callback, as the input/output signature is fixed by `[pyaudio]`. The globals allow algorithm parameters to be available inside the callback, e.g., filter coefficients, and in the case of a digital filter, the filter state must be maintained from frame-to-frame. We will see in the examples section how `scipy.signal.lfilter()`, which implements (1), conveniently supports frame-based digital filtering. To allow interactive control of parameters of the DSP algorithm we can use `ipywidgets`. We will also see later the sliders widgets are particularly suited to this task.

Anatomy of a PyAudio Callback function

Before writing the callback we first need to instantiate a `DSP_io_stream` object, as shown in the following code block:

```
DSP_IO = pah.DSP_io_stream(callback, #callback name
                          2,2, # set I/O device indices
                          fs=48000, # sampling rate
                          Tcapture=0) # capture buffer length
```

The constructor for `DSP_io_stream` of Figure 3 and the code block above confirm that most importantly we need to supply a function callback name, and most likely provide custom input/output device numbers, choose a sampling rate, and optionally choose the length of the capture buffer.

A basic single channel *loop through* callback function, where the input samples are passed to the output, is shown in the code block below:

```
# define a pass through, y = x, callback
def callback(in_data, frame_length, time_info,
            status):
    global DSP_IO, b, a, zi #no widgets yet
    DSP_IO.DSP_callback_tic() #log entering time
    # convert audio byte data to an int16 ndarray
    in_data_ndarray = np.frombuffer(in_data,
                                    dtype=np.int16)

    #*****
    # Begin DSP operations here
    # for this app cast int16 to float32
    x = in_data_ndarray.astype(float32)
    y = x # pass input to output
    # Typically more DSP code here
    # Optionally apply a linear filter to the input
    #y, zi = signal.lfilter(b,a,x,zi=zi)
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples if enabled
    # with Tcapture
    DSP_IO.DSP_capture_add_samples(y)
    #*****
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc() #log departure time
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

The `frame_length` has been set to 1024, and of the four required inputs from [pyaudio], the first, `in_data`, is the input buffer which we first convert to a `int16` ndarray using `np.frombuffer`, and then as a working array convert to `float32`. Note to fill the full dynamic range of the fixed-point signal samples, means that the $x[n]$ sample values can range over $[-2^{15}, 2^{15} - 1]$. Passing over the comments we set $y=x$, and finally convert the output array y back to `int16` and then in the return line back to a byte-string buffer using `.tobytes()`. In general when y is converted from `float` back to `int16`, clipping/overflow will occur unless the dynamic range mentioned above is observed. Along the way code instrumentation methods from Figure 3 are included to record time spent in the callback (`DSP_callback_tic()` and `DSP_callback_toc()`) and store samples for later analysis in the attribute `capture_buffer` (`DSP_capture_add_samples`). These features will be examined in an upcoming example.

To start streaming we need to call the method `interactive_stream()`, which runs the stream in a thread and displays ipywidgets start/stop buttons below the code cell as shown in Figure 5.



Fig. 5: Setting up an interactive stream for the simple $y = x$ loop through, using a run time of 0, which implies run forever.

Performance Measurements

The loop through example is good place to explore some performance metrics of 2, and take a look at some of the instrumentation that is part of the `DSP_io_stream` class. The methods `DSP_callback_tic()` and `DSP_callback_toc()` store time stamps in attributes of the class. Another attribute stores

samples in the attribute `data_capture`. For the instrumentation to collect operating data we need to set `Tcapture` greater than zero. We will also set the total run time to 2s:

```
DSP_IO = pah.DSP_io_stream(callback,2,2,fs=48000,
                          Tcapture=2)
DSP_IO.interactive_stream(2,1)
```

Running the above in Jupyter notebook cell will capture 2s of data. The method `stream_stats()` displays the following:

```
Ideal Callback period = 21.33 (ms)
Average Callback Period = 21.33 (ms)
Average Callback process time = 0.40 (ms)
```

which tells us that as expected for a sampling rate of 48 kHz, and a frame length of 1024 is simply

$$T_{\text{callback period}} = 1024 \times \frac{1}{48000} = 21.33 \text{ ms} \quad (2)$$

The time spent in the callback should be very small, as very little processing is being done. We can also examine the callback latency by first having the AD2 input a low duty cycle pulse train at a 2 Hz rate, thus having 500 ms between pulses. We then use the scope to measure the time difference between the input (scope channel C2) and output (scope channel C1) waveforms. The resulting plot is shown in Figure 6. We see that PyAudio and the PC audio subsystem introduces about 70.7ms of latency. A hybrid iMic ADC and builtin DAC results in 138 ms on macOS. Moving to Win 10 latency increases to 142 ms, using default USB drivers.

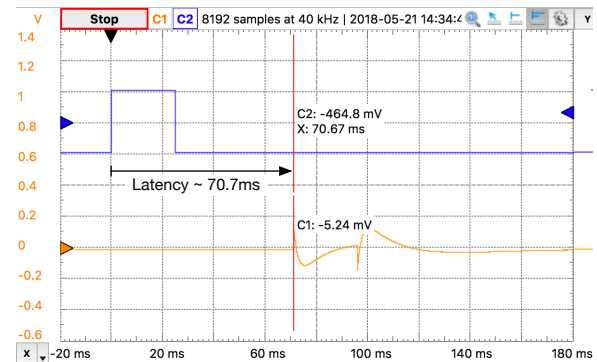


Fig. 6: Callback latency measurement using the AD2 where C2 is the input and C1 is the output, of a 2 Hz pulse train in the loop through app.

The frequency response magnitude of an LTI system can be measured using the fact that [Opp2010] at the output of a system driven by white noise, the measured power output spectrum is a scaled version of the underlying system frequency response magnitude squared, i.e.,

$$S_{y,\text{measured}}(f) = \sigma_x^2 |H_{\text{LTI system}}(f)|^2 \quad (3)$$

where σ_x^2 is the variance of the input white noise signal. Here we use this technique to first estimate the frequency response magnitude of the input path (ADC only) using the attribute `DSP_IO.capture_buffer`, and secondly take end-to-end (ADC-DAC) measurements using the AD2 spectrum analyzer in dB average mode (500 records). In both cases the white noise input is provided by the AD2 function generator. Finally, the AD2 measurement is saved to a CSV file and imported into the Jupyter notebook, as shown in the code block below. This allows

an overlay of the ADC and ADC-DAC measurements, entirely in the Jupyter notebook.

```
import sk_dsp_comm.sigsys as ss
f_AD, Mag_AD = loadtxt('Loop_through_noise_SA.csv',
                    delimiter=',', skiprows=6,
                    unpack=True)
Pxx, F = ss.my_psd(DSP_IO.data_capture, 2**11, 48000);
plot(F, 10*log10(Pxx/Pxx[20]))
plot(f_AD, Mag_AD-Mag_AD[100])
ylim([-10, 5])
xlim([0, 20e3])
ylabel(r'ADC Gain Flatness (dB)')
xlabel(r'Frequency (Hz)')
legend((r'ADC only from DSP_IO.capture_buffer',
        'ADC-DAC from AD2 SA dB Avg'))
title(r'Loop Through Gain Flatness using iMic at
      f_s = 48 kHz')
grid();
savefig('Loop_through_iMic_gain_flatness.pdf')
```

The results are compared in Figure 7, where we see a roll-off of about 3 dB at about 14 kHz in both the ADC path and the composite ADC-DAC path. The composite ADC-DAC begins to rise above 17 kHz and flattens to 2 dB down from 18-20 kHz. As a practical matter, humans do not hear sound much above 16 kHz, so the peaking is not much of an issue. Testing of the Sabrent device the composite ADC-DAC 3 dB roll-off occurs at about 17 kHz. The native PC audio output can for example be tested in combination with the iMic or Sabrent ADCs.

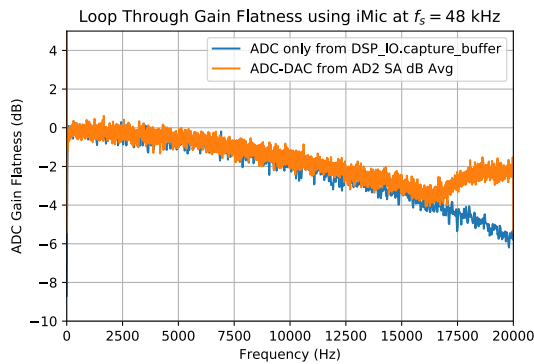


Fig. 7: Gain flatness of the loop through app of just the ADC path via the `DSP_IO.capture_buffer` and then the ADC-DAC path using the AD2 spectrum analyzer to average the noise spectrum.

Examples

In this section we consider a collection of applications examples. This first is a simple two channel loop-through with addition of left and right gain sliders. The second is again two channel, but now cross left-right panning is developed. In of these examples the DSP is memoryless, so there is no need to maintain state using Python globals. The third example is an equal-ripple bandpass filter, which utilizes `sk_dsp_comm.fir_design_helper` to design the filter. The final example develops a three-band audio equalizer using *peaking filters* to raise and lower the gain over a narrow band of frequencies.

Left and Right Gain Sliders

In this first example the signal processing is again minimal, but now two-channel (stereo) processing is utilized, and left and right channel gain slider using `ipywidgets` are introduced. Since

the audio stream is running in a thread, the `ipywidgets` can freely run and interactively control parameters inside the callback function. The two slider widgets are created below, followed by the callback, and finally calling the `interactive_stream` method to run without limit in two channel mode. A 1 kHz sinusoid test signal is input to the left channel and a 5 kHz sinusoid is input to the right channel. While viewing the AD2 scope output in real-time, the gain sliders are adjusted and the signal levels move up and down. A screenshot taken from the Jupyter notebook is combined with a screenshot of the scope output to verify the correlation between the observed signal amplitudes and the slider positions is given in Figure 8. The callback listing, including the set-up of the `ipywidgets` gain sliders, is given below:

```
# Set up two sliders
L_gain = widgets.FloatSlider(description = 'L Gain',
                             continuous_update = True,
                             value = 1.0, min = 0.0,
                             max = 2.0, step = 0.01,
                             orientation = 'vertical')
R_gain = widgets.FloatSlider(description = 'R Gain',
                             continuous_update = True,
                             value = 1.0, min = 0.0,
                             max = 2.0, step = 0.01,
                             orientation = 'vertical')

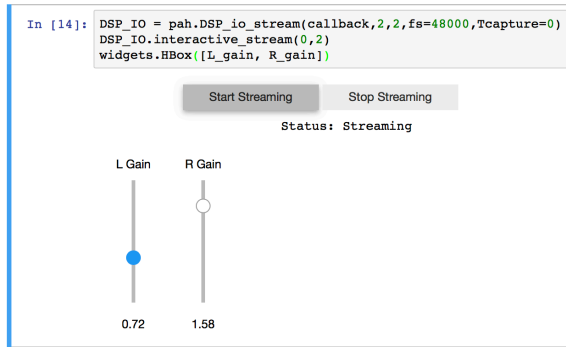
# L and Right Gain Sliders callback
def callback(in_data, frame_count, time_info,
            status):
    global DSP_IO, L_gain, R_gain
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    # separate left and right data
    x_left, x_right = DSP_IO.get_LR(in_data_nda.\
                                    astype(float32))
    #*****
    # DSP operations here
    y_left = x_left*L_gain.value
    y_right = x_right*R_gain.value
    #*****
    # Pack left and right data together
    y = DSP_IO.pack_LR(y_left, y_right)
    # Typically more DSP code here
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples_stereo(y_left,
                                          y_right)
    #*****
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

Note for this two channel stream, the audio subsystem interleaves left and right samples, so now the class methods `get_LR` and `pack_LR` of Figure 3 are utilized to unpack the left and right samples and then repack them, respectively. A screenshot of the gain sliders app, including an AD2 scope capture, with C1 on the left channel and C2 on the right channel, is given in Figure 8.

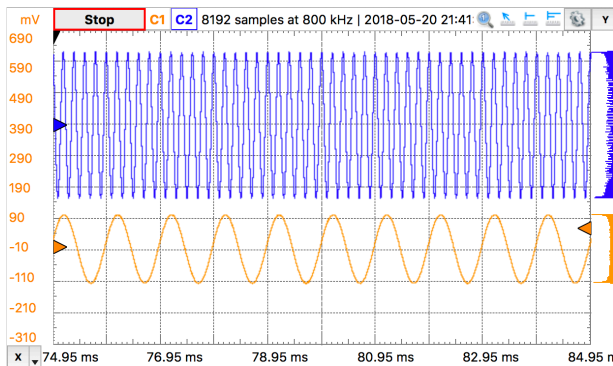
The ability to control the left and right audio level are as expected, especially when listening.

Cross Left-Right Channel Panning

This example again works with a two channel signal flow. The application is to implement a cross channel panning system. Ordinarily panning moves a single channel of audio from 100%



(a) Jupyter notebook start/stop stream controls and left/right gain sliders



(b) Audio outputs for a 1 kHz left input and 5 kHz right input

Fig. 8: A simple stereo gain slider app: (a) Jupyter notebook interface and (b) testing using the AD2 with generators and scope channel C1 (orange) on left and C2 (blue) on right.

left to 100% right as a slider moves from 0% to 100% of its range. At 50% the single channel should have equal amplitude in both channels. In cross channel panning two input channels are super imposed, but such that at 0% the left and right channels are fully in their own channel. At 50% the left and right outputs are equally mixed. At 100% the input channels are now swapped. Assuming that a represents the panning values on the interval $[0, 100]$, a mathematical model of the cross panning app is

$$L_{\text{out}} = (100 - a)/100 \times L_{\text{in}} + a/100 \times R_{\text{in}} \quad (4)$$

$$R_{\text{out}} = a/100 \times L_{\text{in}} + (100 - a)/100 \times R_{\text{in}} \quad (5)$$

where L_{in} and L_{out} are the left channel inputs and outputs respectively, and similarly R_{in} and R_{out} for the right channel. In code we have:

```
# Cross Panning
def callback(in_data, frame_length, time_info,
            status):
    global DSP_IO, panning
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    # separate left and right data
    x_left, x_right = DSP_IO.get_LR(in_data_nda.\
                                    astype(float32))
    #*****
    # DSP operations here
    y_left = (100-panning.value)/100*x_left \
              + panning.value/100*x_right
    y_right = panning.value/100*x_left \
              + (100-panning.value)/100*x_right
    #*****
    # Pack left and right data together
```

```
y = DSP_IO.pack_LR(y_left, y_right)
# Typically more DSP code here
#*****
# Save data for later analysis
# accumulate a new frame of samples
DSP_IO.DSP_capture_add_samples_stereo(y_left,
                                       y_right)
#*****
# Convert from float back to int16
y = y.astype(int16)
DSP_IO.DSP_callback_toc()
# Convert ndarray back to bytes
return y.tobytes(), pah.pyaudio.paContinue
```

This app is best experienced by listening, but visually Figure 9 shows a series of scope captures, parts (b)-(d), to explain how the sounds sources swap from side-to-side as the panning value changes.

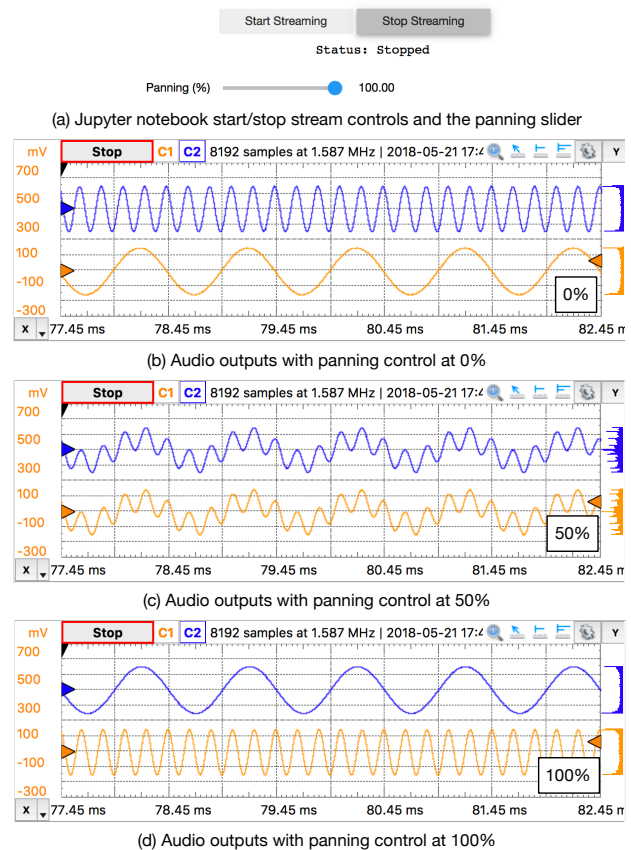


Fig. 9: Cross left/right panning control: (a) launching the app in the Jupyter notebook and (b)-(d) a sequence of scope screenshots as the panning slider is moved from 0% to 50%, and then to 100%.

For dissimilar left and right audio channels, the action of the slider creates a spinning effect when listening. It is possible to extend this app with an automation, so that a low frequency sinusoid or other waveform changes the panning value at a rate controlled by a slider.

FIR Bandpass Filter

In this example we design a high-order FIR bandpass filter using `sk_dsp_comm.fir_design_helper` and then implement the design to operate at $f_s = 48$ kHz. Here we choose the bandpass critical frequencies to be 2700, 3200, 4800, and 5300 Hz, with a passband ripple of 0.5 dB and stopband attenuation of 50 dB (see `fir_d`). Theory is compared with AD2 measurements

using, again using noise excitation. When implementing a digital filter using frame-based processing, `scipy.signal.lfilter` works nicely. The key is to first create a zero initial condition array `zi` and hold this in a global variable. Each time `lfilter` is used in the callback the old initial condition `zi` is passed in, then the returned `zi` is held until the next time through the callback.

```
import sk_dsp_comm.fir_design_helper as fir_d
import scipy.signal as signal
b = fir_d.fir_remez_bpf(2700,3200,4800,5300,
                      .5,50,48000,18)

a = [1]
# Set up a zero initial condition to start
zi = signal.lfiltic(b,a,[0])

# define callback (#2)
def callback2(in_data, frame_length, time_info,
             status):
    global DSP_IO, b, a, zi
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                               dtype=np.int16)

    #*****
    # DSP operations here
    # Here we apply a linear filter to the input
    x = 5*in_data_nda.astype(float32)
    #y = x
    # The filter state/memory, zi,
    # must be maintained from frame-to-frame,
    # so hold it in a global
    # for FIR or simple IIR use:
    y, zi = signal.lfilter(b, a, x, zi=zi)
    # for IIR use second-order sections:
    #y, zi = signal.sosfilt(sos, x, zi=zi)
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples(y)
    #*****
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    return y.tobytes(), pah.pyaudio.paContinue

DSP_IO = pah.DSP_io_stream(callback2,2,2,
                          fs=48000,Tcapture=0)
DSP_IO.interactive_stream(Tsec=0,numChan=1)
```

Following the call to `DSP_io.interactive_stream()` the *start* button is clicked and the AD2 spectrum analyzer estimates the power spectrum. The estimate is saved as a CSV file and brought into the Jupyter notebook to overlay the theoretical design. The comparison results are given in Figure 10.

The theory and measured magnitude response plots are in very close agreement, making the end-to-end design, implement, test very satisfying.

Three Band Equalizer

Here we consider the second-order peaking filter, which has infinite impulse response, and place three of them in cascade with a ipywidgets slider used to control the gain of each filter. The peaking filter is used in the design of audio equalizer, where perhaps each filter is centered on octave frequency spacings running from from 10 Hz up to 16 kHz, or so. Each peaking filter can be implemented as a 2nd-order difference equation, i.e., $N = 2$ in equation (1). The design equations for a single peaking filter are given below using z-transform [Opp2010] notation:

$$H_{pk}(z) = C_{pk} \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (6)$$

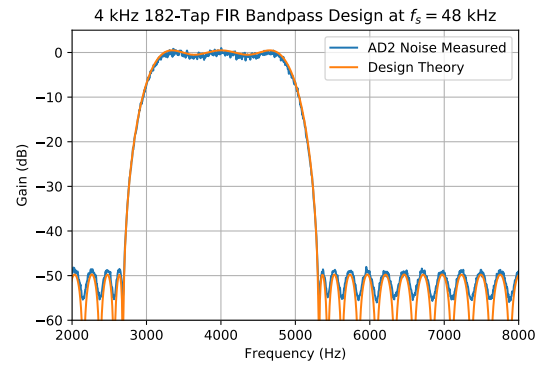


Fig. 10: An overlay plot of the theoretical frequency response with the measured using an AD2 noise spectrum capture import to the Jupyter notebook.

which has coefficients

$$C_{pk} = \frac{1 + k_q \mu}{1 + k_q} \quad (7)$$

$$k_q = \frac{4}{1 + \mu} \tan\left(\frac{2\pi f_c / f_s}{2Q}\right) \quad (8)$$

$$b_1 = \frac{-2 \cos(2\pi f_c / f_s)}{1 + k_q \mu} \quad (9)$$

$$b_2 = \frac{1 - k_q \mu}{1 + k_q \mu} \quad (10)$$

$$a_1 = \frac{-2 \cos(2\pi f_c / f_s)}{1 + k_q} \quad (11)$$

$$a_2 = \frac{1 - k_q}{1 + k_q} \quad (12)$$

where

$$\mu = 10^{G_{dB}/20}, \quad Q \in [2, 10] \quad (13)$$

and f_c is the center frequency in Hz relative to sampling rate f_s in Hz, and G_{dB} is the peaking filter gain in dB. Conveniently, the function peaking is available in the module `sk_dsp_comm.sigsys`. The app code is given below starting with the slider creation:

```
band1 = widgets.FloatSlider(description \
                             = '100 Hz',
                             continuous_update = True,
                             value = 2.0, min = -20.0,
                             max = 20.0, step = 1,
                             orientation = 'vertical')
band2 = widgets.FloatSlider(description \
                             = '1000 Hz',
                             continuous_update = True,
                             value = 10.0, min = -20.0,
                             max = 20.0, step = 1,
                             orientation = 'vertical')
band3 = widgets.FloatSlider(description \
                             = '8000 Hz',
                             continuous_update = True,
                             value = -1.0, min = -20.0,
                             max = 20.0, step = 1,
                             orientation = 'vertical')
```

```
import sk_dsp_comm.sigsys as ss
import scipy.signal as signal
b_b1,a_b1 = ss.peaking(band1.value,100,Q=3.5,
                     fs=48000)
zi_b1 = signal.lfiltic(b_b1,a_b1,[0])
b_b2,a_b2 = ss.peaking(band2.value,1000,Q=3.5,
                     fs=48000)
```

```

zi_b2 = signal.lfilter(b_b2,a_b2,[0])
b_b3,a_b3 = ss.peaking(band3.value,8000,Q=3.5,
                      fs=48000)
zi_b3 = signal.lfilter(b_b3,a_b3,[0])
b_12,a_12 = ss.cascade_filters(b_b1,a_b1,b_b2,a_b2)
b_123,a_123 = ss.cascade_filters(b_12,a_12,b_b3,a_b3)
f = logspace(log10(50),log10(10000),100)
w,H_123 = signal.freqz(b_123,a_123,2*pi*f/48000)
semilogx(f,20*log10(abs(H_123)))
grid();

# define a pass through, y = x, callback
def callback(in_data, frame_length, time_info,
            status):
    global DSP_IO, zi_b1, zi_b2, zi_b3
    global band1, band2, band3
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_ndarray = np.frombuffer(in_data,
                                    dtype=np.int16)
    #*****
    # DSP operations here
    # Here we apply a linear filter to the input
    x = in_data_ndarray.astype(float32)
    #y = x
    # Design the peaking filters on-the-fly
    # and then cascade them
    b_b1,a_b1 = ss.peaking(band1.value,100,
                          Q=3.5,fs=48000)
    z1, zi_b1 = signal.lfilter(b_b1,a_b1,x,
                              zi=zi_b1)
    b_b2,a_b2 = ss.peaking(band2.value,1000,
                          Q=3.5,fs=48000)
    z2, zi_b2 = signal.lfilter(b_b2,a_b2,z1,
                              zi=zi_b2)
    b_b3,a_b3 = ss.peaking(band3.value,8000,
                          Q=3.5,fs=48000)
    y, zi_b3 = signal.lfilter(b_b3,a_b3,z2,
                              zi=zi_b3)
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples(y)
    #*****
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue

```

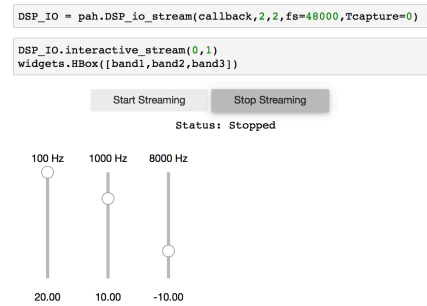
Following the call to `DSP_io.interactive_stream()` the *start* button is clicked and the FFT spectrum analyzer estimates the power spectrum. The estimate is saved as a CSV file and brought into the Jupyter notebook to overlay the theoretical design. The comparison results are given in Figure 11.

Reasonable agreement is achieved, but listening to music is a more effective way of evaluating the end result. To complete the design more peaking filters should be added.

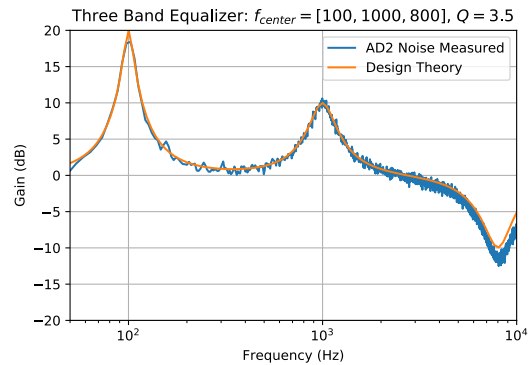
Conclusions and Future Work

In this paper we have described an approach to implement real-time DSP in the Jupyter notebook. This real-time capability rests on top of PyAudio and the wrapper class `DSP_io_stream` contained in `sk_dsp_comm.pyaudio_helper`. The `ipywidgets` allow for interactivity while real-time DSP code is running. The `callback` function does the work using frame-based algorithms, which takes some getting used to. By working through examples we have shown that much can be accomplished with little coding.

A limitation of using PyAudio is the input-to-output latency. At a 48 kHz sampling rate a simple loop through app has around



(a) Jupyter notebook start/stop stream controls and peaking gain sliders



(b) Composite three band frequency response; theory and noise spectrum

Fig. 11: Three band equalizer: (a) launching the app in the Jupyter notebook and (b) an overlay plot of the theoretical log-frequency response with the measured using an AD2 noise spectrum capture import to the Jupyter notebook.

70 ms of delay. For the application discussed in the paper latency is not a show stopper.

In the future we hope to easily develop algorithms that can demodulate software-defined radio (SDR) streams and send the recovered modulation signal out the computer's audio interface via PyAudio. Environments such as GNURadio companion already support this, but being able to do this right in the Jupyter notebook is our desire.

REFERENCES

- [cortexM4] *The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors*. (2016, November). Retrieved June 25, 2018, from <https://community.arm.com/processors/b/blog/posts/white-paper-dsp-capabilities-of-cortex-m4-and-cortex-m7>.
- [Scipysignal] *Signal Processing*. (2018, May 5). Retrieved June 25, 2018 from <https://docs.scipy.org/doc/scipy/reference/signal.html>.
- [DSPComm] *scikit-dsp-comm*. (2018, June 22). Retrieved June 25, 2018 from <https://github.com/mwickert/scikit-dsp-comm>.
- [pyaudio] *PyAudio*, (2017, March). Retrieved June 25, 2018, from <https://people.csail.mit.edu/hubert/pyaudio/>.
- [portaudio] *Port Audio*. (2012, January 25). Retrieved June 25, 2018 from <http://www.portaudio.com/>.
- [ipywidgets] *ipywidgets*. (2018, June 11). Retrieved June 25, 2018, from <https://github.com/jupyter-widgets/ipywidgets>.
- [Opp2010] Oppenheim, A and Schaffer, R (2010). *Discrete-Time Signal Processing* (3rd ed.). New Jersey: Prentice Hall.
- [AD2] *Analog Discovery 2*. (2018, June). Retrieved June 25, 2018 from <https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/>.
- [3p5mm] *3.5mm Analog Discovery Adaptor Design*. (2018, January 30). Retrieved June 25, 2018 from <http://www.eas.uccs.edu/~mwickert/ece5655/>.

Organic Molecules in Space: Insights from the NASA Ames Molecular Database in the era of the James Webb Space Telescope

Matthew J. Shannon^{‡§*}, Christian Boersma^{¶§}

Abstract—We present the software tool pyPAHdb to the scientific astronomical community, which is used to characterize emission from one of the most prevalent types of organic molecules in space, namely polycyclic aromatic hydrocarbons (PAHs). It leverages the detailed studies of organic molecules done at the NASA Ames Research Center. pyPAHdb is a streamlined Python version of the NASA Ames PAH IR Spectroscopic Database (PAHdb; www.astrochemistry.org/pahdb) suite of IDL tools. PAHdb has been extensively used to analyze and interpret the PAH signature from a plethora of emission sources, ranging from solar-system objects to entire galaxies. pyPAHdb decomposes astronomical PAH emission spectra into contributing PAH sub-classes in terms of charge and size using a database-fitting technique. The inputs for the fit are spectra constructed using the spectroscopic libraries of PAHdb and take into account the detailed photo-physics of the PAH excitation/emission process.

Index Terms—astronomy, databases, fitting, data analysis

Science rationale

Polycyclic aromatic hydrocarbons

Polycyclic aromatic hydrocarbons (PAHs) are a class of molecules found throughout the Universe that drive many critical astrophysical processes. They dominate the mid-infrared (IR) emission of many astronomical objects, as they absorb ultraviolet (UV) photons and re-emit that energy through a series of IR emission features between 3-20 μm . They are seen in reflection nebulae, protoplanetary disks, the diffuse interstellar medium (ISM), planetary nebulae, and entire galaxies (e.g., Figure 1), among other environments. Structurally, they are composed of a hexagonal carbon lattice (see Figure 2); taken as an entire family, they are by far the largest known molecules in space. PAHs are exceptionally stable, allowing them to survive the harsh conditions amongst a remarkably wide variety of astronomical objects.

The role of astronomical PAHs

Thanks to their ubiquity, PAH IR emission signatures are routinely used by astronomers as probes of object type and astrophysical

* Corresponding author: Matthew.J.Shannon@nasa.gov

‡ Universities Space Research Association, Columbia, MD

§ NASA Ames Research Center, MS245-6, Moffett Field, CA 94035-1000

¶ San José State University Research Foundation, 210 N 4th St Fl 4, San Jose, CA 95112

Copyright © 2018 Matthew J. Shannon et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.



Fig. 1: A combined visible light-IR image from the Spitzer Space Telescope of the galaxy Messier-82 (M82), also known as the Cigar galaxy because of its cigar-like shape in visible light. The red region streaming away from the galaxy into intergalactic space traces the IR emission from PAHs. Credits: NASA/JPL-Caltech/C. Engelbracht (Steward Observatory) and the SINGS team.

processes. For example, the PAH IR signature is used as an indicator of star formation in high redshift galaxies [RPD⁺14] and to differentiate between black hole and starburst engines in galactic nuclei [GLS⁺98]. Those astronomers who study star and planet formation use the IR PAH signature as an indicator of the geometry of circumstellar disks [MWB⁺01] [BPM⁺09].

PAHs are believed to form in the circumstellar ejecta of late-type stars, after which they become part of the ISM as the material travels away from the star. Over time, PAHs are incorporated into dense clouds, wherein they participate in ongoing chemistry and are eventually brought into newly-forming star and budding planetary systems.

They play important roles in circumstellar processes and the diffuse ISM by modulating radiation fields and influencing charge balance. Once incorporated into dense molecular clouds, they can dominate cloud cooling and promote H₂ formation. PAHs also control the large-scale ionization balance and thereby the coupling

of magnetic fields to the gas. Through their influence on the forces supporting clouds against gravity, PAHs also affect the process of star formation itself. They are a major contributor to the heating of diffuse atomic gas in the ISM and thereby the physical conditions in such environments and its structure.

The unique properties of PAHs, coupled with their spectroscopic response to changing astrophysical conditions and their ability to convert UV photons to IR radiation, makes them powerful probes of astronomical objects at all stages of the stellar life cycle. Notably, they allow astronomers to probe properties of diffuse media in regions not normally accessible.

NASA Ames PAH IR Spectroscopic Database (PAHdb)

The Astrophysics & Astrochemistry Laboratory at NASA Ames Research Center [NAS] provides data and tools for analyzing and interpreting astronomical PAH spectra. The NASA Ames PAH IR Spectroscopic Database (PAHdb; [BRBA18] [BBR⁺14]) is the culmination of more than 30 years of laboratory and computational research carried out at the NASA Ames Research Center to test and refine the astronomical PAH model. PAHdb consists of three components (all under the moniker of "PAHdb"): the spectroscopic libraries, the website (see Figure 2), and the suite of off-line IDL¹ tools. PAHdb has the world's foremost collection of PAH spectra.

PAHdb is highly cited and is used to characterize and understand organic molecules in our own Galaxy and external galaxies. The database includes a set of innovative astronomical models and tools that enables astronomers to probe and quantitatively analyze the state of the PAH population. For instance, one can derive PAH ionization balance, size, structure, and composition and tie these to the prevailing local astrophysical conditions (e.g., electron density, parameters of the radiation field, etc.) [BBA16] [BBA18].

NASA's next great observatory for PAH research: JWST

The next great leap forward for IR astronomy is the the James Webb Space Telescope (JWST). JWST is NASA's next flagship observatory and the successor to the exceptionally successful *Hubble Space Telescope* (www.nasa.gov/hubble) and *Spitzer Space Telescope* (www.nasa.gov/spitzer). JWST is being developed through a collaboration between NASA, the European Space Agency (ESA) and the Canadian Space Agency (CSA). The telescope features a primary mirror with a diameter of 6.5 m and carries four science instruments. These instruments will observe the Universe with unprecedented resolution and sensitivity in the near- and mid-IR. The observatory is expected to launch early 2021.

As part of an awarded JWST Early Release Science (ERS) program², we are developing a Python-based toolkit for quickly analyzing PAH emission in IR spectroscopic data

pyPAHdb: a tool designed for JWST

The purpose of pyPAHdb is to derive astronomical parameters directly from JWST observations, but the tool is not limited to JWST observations alone. pyPAHdb is the light version of a full suite of Python software tools³ that is currently being developed, which is an analog of the off-line IDL tools⁴. A feature comparison is made in Table 1 (see also Section "The underlying PAH photo-physics"). pyPAHdb will enable PAH experts and non-experts

1. IDL is a registered trademark of Harris Geospatial.

2. The ERS program is titled "Radiative Feedback from Massive Stars as Traced by Multiband Imaging and Spectroscopic Mosaics" (jwst-ism.org; ID: 1288).

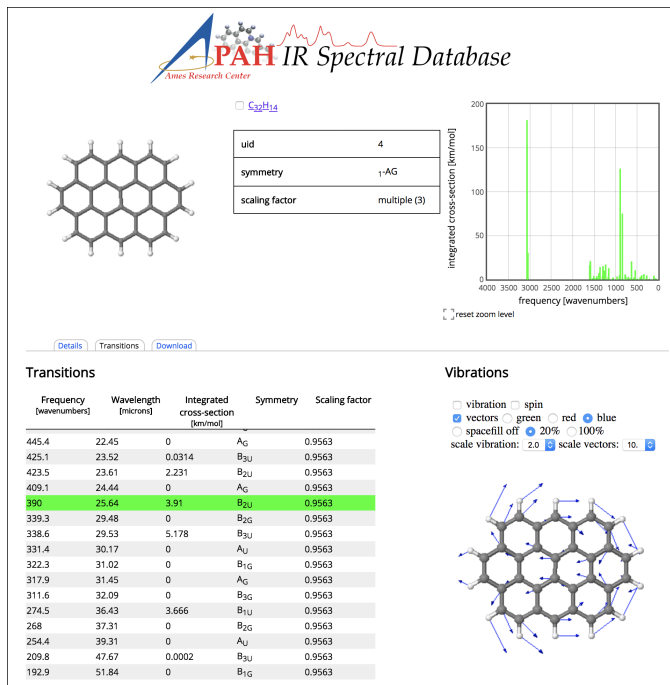


Fig. 2: Screenshot of the NASA Ames PAH IR Spectroscopic Database website located at www.astrochemistry.org/pahdb/. Shown here are the details and vibrational spectrum for the PAH molecule ovalene (C₃₂H₁₄). Additionally, each vibrational transition is animated and can be inspected for ease of interpretation (shown in the lower-right).

	pyPAHdb	IDL/Python tools
Included molecules.	Fixed	User defined
Excitation energy	Fixed	User defined
Emission profile	Fixed	Selectable
FWHM	Fixed	User defined
Band redshift	Fixed	User defined
Emission model	Fixed	Selectable
NNLS	✓	✓
Class breakdown	✓	✓
Parallelization	✓	✓
Handle uncertainties		✓

TABLE 1: Feature comparison between pyPAHdb and the full suites of off-line IDL/Python tools. Note; NNLS is non-negative least squares; FWHM is full-width at half-maximum of an emission profile; "uncertainties" in this context refers to handling observational spectroscopic uncertainties.

alike to analyze and interpret astronomical PAH emission spectra.

pyPAHdb analyzes spectroscopic observations (including spectral maps) and characterizes the PAH emission using a database-fitting approach, providing the PAH ionization and size fractions.

The package is imported using the following statement:

```
import pypahdb
```

3. AmesPAHdbPythonSuite: github.com/PAHdb/AmesPAHdbPythonSuite

4. AmesPAHdbIDLSuite: github.com/PAHdb/AmesPAHdbIDLSuite

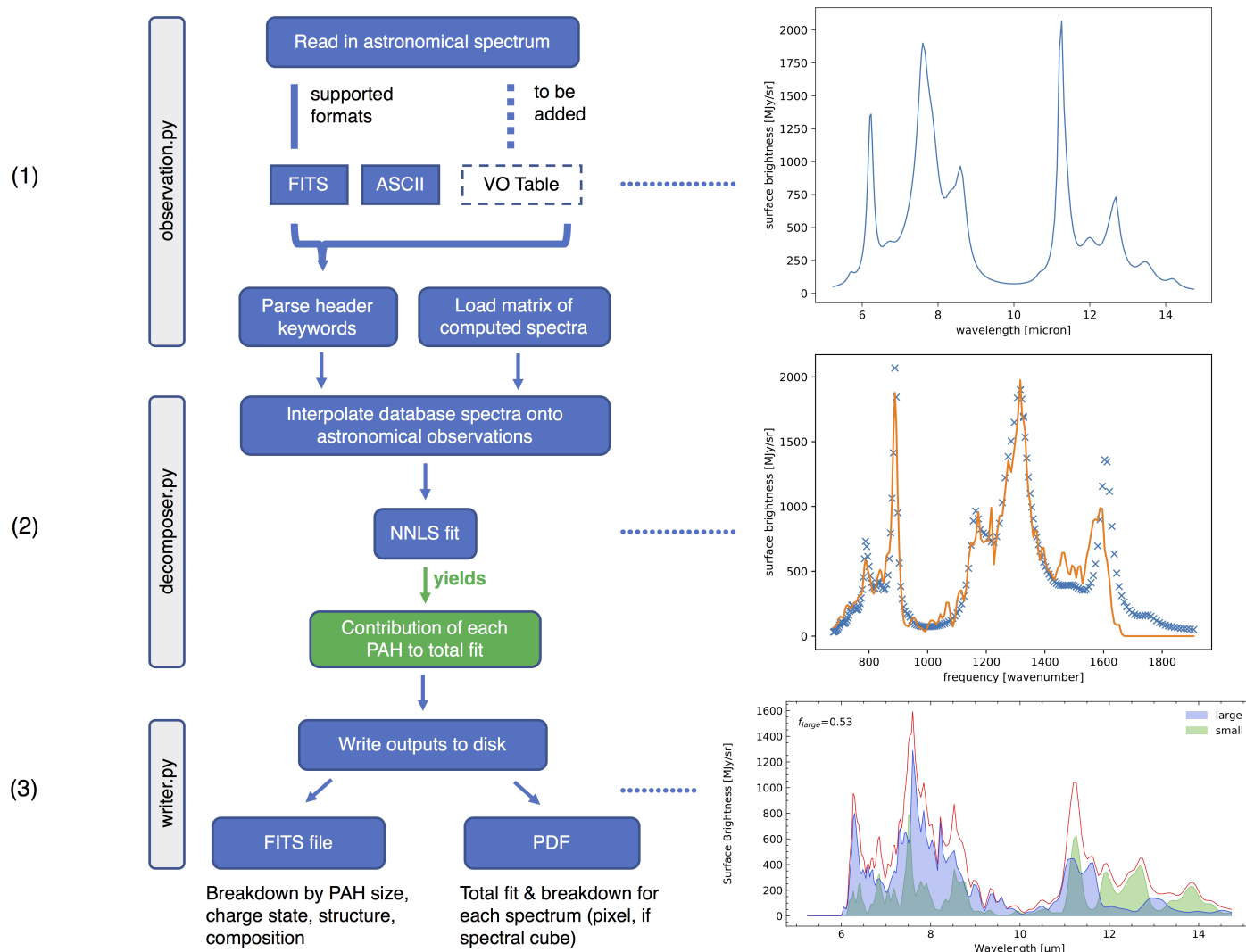


Fig. 3: *pyPAHdb* flowchart. (1) Astronomical spectroscopic data is loaded, whether represented in FITS or ASCII files. (2) An over-sampled pre-computed matrix of PAH spectra is loaded and interpolated onto the wavelength grid of the astronomical observations. Database-fitting is performed using non-negative least-squares (NNLS), which yields the contribution of an individual PAH molecule to the total fit. As a result, we obtain a breakdown of the model fit in terms of PAH charge and size. (3) The results are written to disk as a single FITS file and a PDF summarizing the model fit (one page per pixel, if a spectral cube is provided as input).

The general program methodology is encapsulated in the flowchart presented in Figure 3 and is as follows:

- (1) Read-in a file containing spectroscopic PAH observations of an astronomical object. This functionality is provided by the class `observation`, which is implemented in `observation.py`. It is the responsibility of the user to ensure all non-PAH emission components have been removed from the spectrum. The class uses a fall-through try-except chain to attempt to read the given filename using the facilities provided by `astropy.io`. The spectroscopic data is stored as a class attribute as a `spectrum` object, which holds the data in terms of abscissa and ordinate values using `numpy` arrays. The units associated with the abscissa and ordinate values are, in the case of a FITS file, determined from the accompanying header, which itself is also stored as a class attribute. The spectral coordinate system is interpreted

from FITS header keywords following the specification by [GCVA06]. The `spectrum` class is implemented in `spectrum.py` and provides functionality to convert between different coordinate representations. Below is example Python code demonstrating the use of the class. The file `NGC7023-NW-BRIGHT.txt_pahs.txt` in this demonstration can be found in the `examples` directory that is part of the `pyPAHdb` package. The output of the following code-block is shown in Figure 3.

```
import pypahdb as pah
import matplotlib.pyplot as plt
file = 'NGC7023-NW-BRIGHT.txt_pahs.txt'
obs = pah.observation(file)
s = obs.spectrum
plt.plot(s.abscissa, s.ordinate[:,0,0])
plt.ylabel(s.units['ordinate']['str'])
plt.xlabel(s.units['abscissa']['str'])
plt.show()
```

- (2) Decompose the observed PAH emission into contributions from different PAH subclasses, here charge and size. This functionality is provided by the class `decomposer`, which is implemented in `decomposer.py`. The class takes as input a `spectrum` object, of which it creates a deep copy and calls its `spectrum.convertunits` method to convert the abscissa units to wavenumber. Subsequently, a pre-computed numpy matrix of highly oversampled PAH emission spectra stored as a `pickle` is loaded from file. Utilizing `numpy.interp`, each of the PAH emission spectra, represented by a single column in the pre-computed matrix, is interpolated onto the frequency grid (in wavenumber) of the input spectrum. This process is parallelized using the `multiprocessing` package. `optimize.nnls` is used to perform a non-negative least-squares (NNLS) fit of the pre-computed spectra to the input spectra. NNLS is chosen because it is appropriate to the problem, fast, and always converges. The solution vector (weights) is stored as an attribute and considered private. Combining lazy instantiation and Python's `@property`, the results of the fit and the breakdown can be retrieved. In case the input spectrum represents a spectral cube and where possible, the calculations are parallelized across each pixel using, again, the `multiprocessing` package. Below is example code demonstrating the use of the class and extends the previous code-block. The output of the code-block is shown in Figure 3.

```
result = pah.decomposer(obs.spectrum)
s = result.spectrum
plt.plot(s.abscissa, s.ordinate[:,0,0], 'x')
plt.ylabel(s.units['ordinate']['str']);
plt.xlabel(s.units['abscissa']['str']);
plt.plot(s.abscissa, result.fit[:,0,0])
plt.show()
```

- (3) Produce output to file given a `decomposer` object. This functionality is provided by the class `writer`, which is implemented in `writer.py`, and serves to summarize the results from the `decomposer` class so that a user may assess the quality of the fit and store the PAH characteristics of their astronomical observations. The class uses `astropy.fits` to write the PAH characteristics to a FITS file and the `matplotlib` package to generate a PDF summarizing the results. The class will attempt to incorporate relevant information from any (FITS) header provided. Below is example code demonstrating the use of the class, which extends the previous code-block. The size breakdown part of the generated PDF output is shown in Figure 3.

```
pah.writer(result, header=obs.header)
```

It is anticipated that pyPAHdb will constitute an effective and useful tool of an astronomer's toolbox, handling thousands of spectra. Therefore, performance is of importance. To measure performance, a spectral cube containing the PAH emission spectra at some 210 pixel locations is analyzed with pyPAHdb (see also Section "Demonstration"). To put the measurement in context, it is compared to analyzing the same spectral cube using the off-line IDL tools. In this comparison the analysis with pyPAHdb is 15 times faster at four seconds as compared to using the IDL tools,

when tested on a 2.8 GHz Intel Core i7 MacBook Pro with 16 GB of memory.

The underlying PAH photo-physics

To analyze astronomical PAH emission spectra with the *absorption* data contained in PAHdb's libraries, the PAHdb data need to be turned into emission spectra. As discussed in the previous section, pyPAHdb hides the underlying photo-physics in a pre-computed matrix that is read-in by the `decomposer` class. The pre-computed matrix is constructed using the full Python suite and takes modeled, highly-over-sampled PAH emission spectra from version 3.00 of the library of computed spectra.

This matrix uses the data on a collection of "astronomical" PAHs, which include those PAHs that have more than 20 carbon atoms, have no hetero-atom substitutions except for possibly nitrogen, have no aliphatic side groups, and are not fully dehydrogenated. In addition, the fullerenes C_{60} and C_{70} are added.

While several more sophisticated emission models are available in the full Python suite, here a PAH's emission spectrum is calculated from the vibrational temperature it reaches after absorbing a single 7 eV photon and making use of the thermal approximation (e.g., [STA93] and [VPM+01]). Table 1 highlights some of the differences between pyPAHdb and the full suite of IDL/Python tools.

The spectral intensity $I_j(\nu)$, in $\text{erg s}^{-1} \text{cm}^{-1} \text{mol}^{-1}$, from a mol of the j^{th} PAH is thus calculated as:

$$I_j(\nu) = \sum_{i=1}^n \frac{2hc\nu_i^3 \sigma_i}{e^{\frac{h\nu_i}{kT}} - 1} \phi(\nu), \quad (1)$$

with ν the frequency in cm^{-1} , h Planck's constant in erg s , c the speed-of-light in cm s^{-1} , ν_i the frequency of mode i in cm^{-1} , σ_i the integrated absorption cross-section for mode i in cm mol^{-1} , k Boltzmann's constant in erg K^{-1} , T the vibrational temperature in K, and $\phi(\nu)$ is the frequency dependent emission profile in cm . The sum is taken over all n modes and the emission profile is assumed Gaussian with a full-width at half-maximum (FWHM) of 15 cm^{-1} . Note that before applying the emission profile, a redshift of 15 cm^{-1} is applied to each of the band positions (ν_i) to mimic some anharmonic effects. This redshift value is currently the best estimate from laboratory experiments (see e.g., the discussion in [BBA13]).

The vibrational temperature attained after absorbing a single 7 eV photon is calculated by the molecule's heat capacity. The heat capacity, C_V in erg K , of a molecular system can be described in terms of isolated harmonic oscillators by:

$$C_V = k \int_0^{\infty} e^{-\frac{h\nu}{kT}} \left[\frac{\frac{h\nu}{kT}}{1 - e^{-\frac{h\nu}{kT}}} \right]^2 g(\nu) d\nu, \quad (2)$$

where $g(\nu)$ is known as the density of states and describes the distribution of vibrational modes. However due to the discrete nature of the modes, the density of states is just a sum of δ -functions:

$$g(\nu) = \sum_{i=1}^n \delta(\nu - \nu_i). \quad (3)$$

The vibrational temperature is ultimately calculated by solving:

$$\int_0^{T_{\text{vibration}}} C_V dT = E_{\text{in}}, \quad (4)$$

where E_{in} is the energy of the absorbed photon—here this is 7 eV.

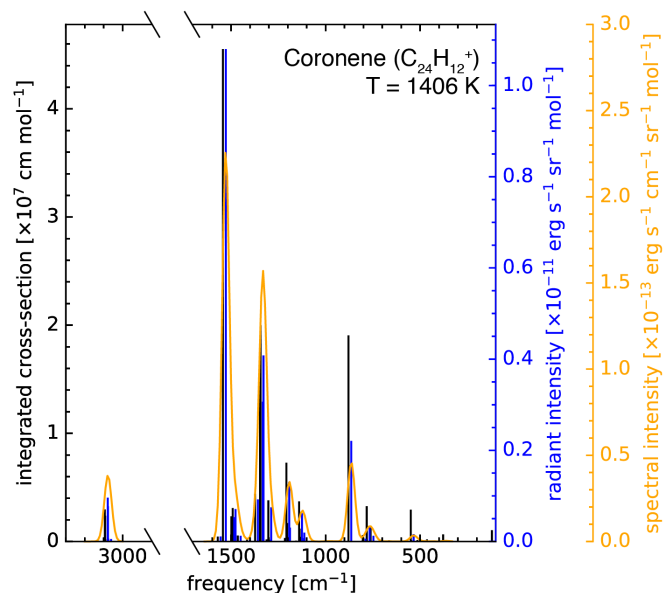


Fig. 4: Demonstration of applying the simple PAH emission model as outlined in Equations 1-4 to the 0 K spectrum of coronene (in black; $C_{24}H_{12}^+$) from version 3.00 of the library of computed spectra of PAHdb. After applying the PAH emission model, but before the convolution with the emission profile, the blue spectrum is obtained. The final spectrum is shown in orange. For display purposes the profiles have been given a FWHM of 45 cm^{-1} .

In Python, in the full suite, Equation 4 is solved using root-finding with `scipy.optimize.brentq`. The integral is calculated with `scipy.optimize.quad`.

Figure 4 illustrates the process on the spectrum of the coronene cation ($C_{24}H_{12}^+$), which reaches a vibrational temperature of 1406 K after absorbing a single 7 eV photon.

Demonstration

As a more sophisticated demonstration of pyPAHdb's utility, we analyze a spectral cube dataset of the reflection nebula NGC 7023, as constructed from *Spitzer Space Telescope* observations. This data cube is overlaid on a visible-light image of NGC 7023 from the *Hubble Space Telescope* in Figure 5, left panel [BBA18].

The spectral cube is aligned such that, in these observations, we observe the transition from diffuse, ionized/atomic species (e.g., HI) near the exciting star to dense, molecular material (e.g., H_2) more distant from the star. The transition zone between the two is the photodissociation region, where PAHs have a strong presence. The properties of the PAH molecules are known to vary across these boundaries, since they are exposed to harsh radiation in the exposed cavity of the diffuse zone, and shielded in the molecular region.

We use pyPAHdb to derive the variability of PAH properties across this boundary layer by analyzing the full spectrum at every pixel. The code-block below, which is taken from `example.py` included in the pyPAHdb distribution, demonstrates how this is done. Note that this is the same general syntax as is used for analyzing a single spectrum, but here `NGC7023.fits` is a spectral cube.

```
# ----- Running pyPAHdb ----- #
# ----- Running pyPAHdb ----- #
# ----- Running pyPAHdb ----- #
```

```
import pyPAHdb
observation = pyPAHdb.observation('NGC7023.fits')
result = pyPAHdb.decomposer(observation.spectrum)

# This will output the results file,
# 'NGC7023_pypahdb.fits':
pyPAHdb.writer(result, header=observation.header)
```

With the results from the entire spectral cube, maps of relevant astrophysical quantities can be constructed. For example, Figure 5 (right panel) presents a map of the varying PAH ionization fraction across NGC 7023. As expected, the fraction is systematically higher across the diffuse region, where PAHs are more exposed to the star, than the dense region, where PAHs are partially shielded from the star. This figure was constructed in the following manner:

```
# ----- Plotting a map of ionization ----- #
# ----- Plotting a map of ionization ----- #
# ----- Plotting a map of ionization ----- #

# Import needed/useful modules.
import matplotlib.pyplot as plt
import numpy as np
from astropy.io import fits
from mpl_toolkits.axes_grid1 import \
    make_axes_locatable

# Read in the results from pyPAHdb.
# The data is 3-dimensional, with the first axis
# denoting the PAH properties, and the latter two
# being spatial.
hdulist = fits.open('NGC7023_pypahdb.fits')
ionization_fraction, large_fraction, norm = \
    hdulist[0].data

# Create a figure instance.
fig = plt.figure()
ax = fig.add_subplot(111)

# Plot our ionization map; we've flipped it left-right
# to match the Hubble image's orientation.
im = ax.imshow(np.fliplr(ionization_fraction),
               origin='upper', cmap='viridis',
               interpolation='nearest')

# Add a nice colorbar.
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%",
                          pad=0.05)
cbar = plt.colorbar(im, cax=cax)
cbar.set_label('ionization fraction [#]',
               rotation=270, labelpad=18)

# Set axes labels.
ax.set_xlabel('pixel [#]')
ax.set_ylabel('pixel [#]')

# Save the figure.
plt.savefig('ionization_fraction_map.pdf',
           format='pdf', bbox_inches='tight')
plt.close()
```

The type of analysis demonstrated here allows users to quickly interpret the distribution of PAHs in their astronomical observations and variations in PAH charge and size. Note that in addition to the ionization fraction, the pyPAHdb results file `NGC7023_pypahdb.fits` contains a data array for the large PAH fraction and norm (accessed and plotted in the same manner), which we have defined in the code above.

Summary

The data and tools provided through PAHdb have proven to be valuable assets for the astronomical community for analyzing and

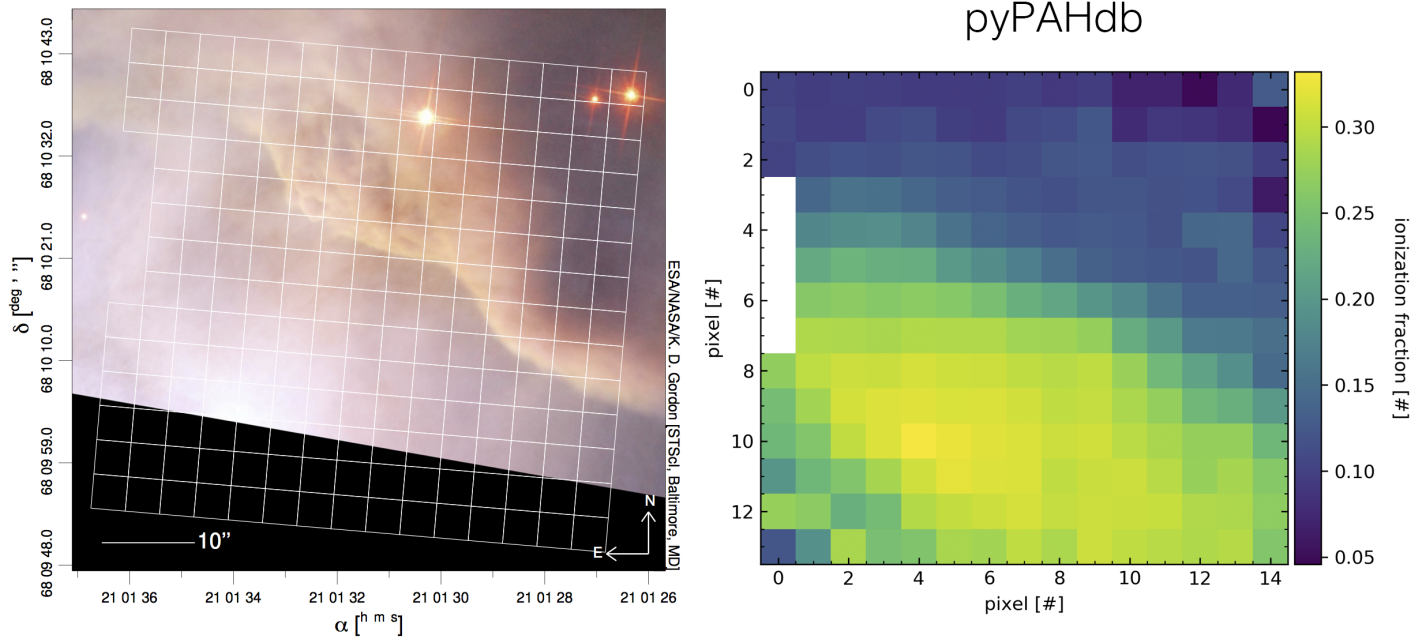


Fig. 5: *Left:* An image of the reflection nebula NGC 7023 as obtained by the Hubble Space Telescope. Overlaid is a pixel grid representing a spectral cube of observations taken with the Spitzer Space Telescope; each pixel contains an infrared spectrum. In this figure, the exciting star is just beyond the lower left corner. We are observing a photodissociation region boundary: the material in the lower half of the figure is diffuse and exposed to the star; the material in the upper (right) half is molecular and shielded from the star. The diagonal boundary separating the two zones is clearly visible. PAHs are common in these environments. Figure adapted from [BBA18]. *Right:* We display PAH ionization across the NGC 7023 (white grid in left panel), using pyPAHdb. Here, an ionization fraction of 1 means all PAHs are ionized, while 0 means all are neutral. Note that in the diffuse, exposed cavity (lower half) the PAHs are on average more ionized than in the denser molecular zone (upper half).

interpreting PAH emission spectra. The launch of *JWST* in 2021 will usher a new era of astronomical PAH research. In the context of an awarded *JWST* Early Release Science program, we are developing pyPAHdb as a key data analysis tool to facilitate quick and effective analysis of PAH emission spectra. While this tool is being developed with *JWST* in mind, it is not limited to *JWST* data: it currently supports spectra from *Spitzer Space Telescope*, *Infrared Space Observatory*, and any user-defined spectrum. pyPAHdb is in active development and will be finalized well before *JWST*'s launch.

REFERENCES

- [BBA13] C. Boersma, J. D. Bregman, and L. J. Allamandola. Properties of Polycyclic Aromatic Hydrocarbons in the Northwest Photon Dominated Region of NGC 7023. I. PAH Size, Charge, Composition, and Structure Distribution. *ApJ*, 769:117, June 2013. doi:10.1088/0004-637X/769/2/117.
- [BBA16] C. Boersma, J. Bregman, and L. J. Allamandola. The Charge State of Polycyclic Aromatic Hydrocarbons Across Reflection Nebulae: PAH Charge Balance and Calibration. *ApJ*, 832:51, November 2016. doi:10.3847/0004-637X/832/1/51.
- [BBA18] C. Boersma, J. Bregman, and L. J. Allamandola. The Charge State of Polycyclic Aromatic Hydrocarbons across a Reflection Nebula, an H II Region, and a Planetary Nebula. *ApJ*, 858:67, May 2018. doi:10.3847/1538-4357/aabcbe.
- [BBR⁺14] C. Boersma, C. W. Bauschlicher, A. Ricca, A. L. Mattioda, J. Cami, E. Peeters, F. Sánchez de Armas, G. Puerta Saborido, D. M. Hudgins, and L. J. Allamandola. The NASA Ames PAH IR Spectroscopic Database Version 2.00: Updated Content, Web Site, and On(Off)line Tools. *ApJS*, 211:8, 2014. doi:10.1088/0067-0049/211/1/8.
- [BPM⁺09] C. Boersma, E. Peeters, N. L. Martín-Hernández, G. van der Wolk, A. P. Verhoeff, A. G. G. M. Tielens, L. B. F. M. Waters, and J. W. Pel. A spatial study of the mid-IR emission features in four Herbig Ae/Be stars. *A&A*, 502:175–187, July 2009. doi:10.1051/0004-6361/200911820.
- [BRBA18] C. W. Bauschlicher, Jr., A. Ricca, C. Boersma, and L. J. Allamandola. The NASA Ames PAH IR Spectroscopic Database: Computational Version 3.00 with Updated Content and the Introduction of Multiple Scaling Factors. *ApJS*, 234:32, February 2018. doi:10.3847/1538-4365/aaa019.
- [GCVA06] E. W. Greisen, M. R. Calabretta, F. G. Valdes, and S. L. Allen. Representations of spectral coordinates in FITS. *A&A*, 446:747–771, February 2006. doi:10.1051/0004-6361:20053818.
- [GLS⁺98] R. Genzel, D. Lutz, E. Sturm, E. Egami, D. Kunze, A. F. M. Moorwood, D. Rigopoulou, H. W. W. Spoon, A. Sternberg, L. E. Tacconi-Garman, L. Tacconi, and N. Thatte. What Powers Ultraluminous IRAS Galaxies? *ApJ*, 498:579, May 1998. doi:10.1086/305576.
- [MWB⁺01] G. Meeus, L. B. F. M. Waters, J. Bouwman, M. E. van den Ancker, C. Waelkens, and K. Malfait. ISO spectroscopy of circumstellar dust in 14 Herbig Ae/Be systems: Towards an understanding of dust processing. *A&A*, 365:476, January 2001. doi:10.1051/0004-6361:20000144.
- [NAS] NASA Ames Research Center. Astrochemistry and astrochemistry laboratory. URL: www.astrochemistry.org.
- [RPD⁺14] D. A. Riechers, A. Pope, E. Daddi, L. Armus, C. L. Carilli, F. Walter, J. Hodge, R.-R. Chary, G. E. Morrison, M. Dickinson, H. Dannerbauer, and D. Elbaz. Polycyclic Aromatic Hydrocarbon and Mid-Infrared Continuum Emission in a $z > 4$ Submillimeter Galaxy. *ApJ*, 786:31, May 2014. doi:10.1088/0004-637X/786/1/31.
- [STA93] W. A. Schutte, A. G. G. M. Tielens, and L. J. Allamandola. Theoretical modeling of the infrared fluorescence from interstellar polycyclic aromatic hydrocarbons. *apj*, 415:397, September 1993. doi:10.1086/173173.
- [VPM⁺01] L. Verstraete, C. Pech, C. Moutou, K. Sellgren, C. M. Wright,

M. Giard, A. Léger, R. Timmermann, and S. Drapatz. The Aromatic Infrared Bands as seen by ISO-SWS: Probing the PAH model. *aa*, 372:981, June 2001. doi:[10.1051/0004-6361:20010515](https://doi.org/10.1051/0004-6361:20010515).

Harnessing the Power of Scientific Python to Investigate Biogeochemistry and Metaproteomes of the Central Pacific Ocean

Noelle A. Held^{§‡}, Jaclyn K. Saunders^{‡§}, Joe Futrelle[‡], Mak A. Saito^{‡*}

<https://youtu.be/WYmAu0GiSU4>



Abstract—Oceanographic expeditions commonly generate millions of data points for various chemical, biological, and physical features, all in different formats. Scientific Python tools are extremely useful for synthesizing this data to make sense of major trends in the changing ocean environment. In this paper, we present our application of scientific Python to investigate metaproteome data from the oxygen-depleted Central Pacific Ocean. The microbial proteins of this region are major drivers of biogeochemical cycles, and represent a living proxy of the ancient anoxic ocean. They also provide a look into the trajectory of the ocean in the face of rising temperatures, which cause deoxygenation. We assessed 103 metaproteome samples collected in the Central Pacific Ocean on the 2016 ProteOMZ cruise. This data represents ~60,000 identified proteins and over 6 million datapoints, in addition to over 6,600 corresponding chemical, physical, and biological metadata points.

An interactive data analysis tool which enables the scientific user to visualize and interrogate patterns in these large metaproteomic datasets in conjunction with hydrographic features was not previously available. Bench scientists who would like to use this oceanographic data to gain insight into marine biogeochemical cycles were at a disadvantage as no tool existed to query these complex datasets in a visually meaningful way. Our goal was to provide a graphical visualization tool to enhance the exploration of these complex dataset; specifically, using interactive tools to enable users the ability to filter and automatically generate plots from slices of large metaproteomic and hydrographic datasets. We developed a Bokeh application [BOKEH] for data exploration which allows the user to hone in on proteins of interest using widgets. The user can then explore relationships between protein abundance and water column depth, hydrographic data, and taxonomic origin. The result is a complete and interactive visualization tool for interrogating a multivariate oceanographic dataset, which helped us to demonstrate a strong relationship between chemical, physical, and biological variables and the microbial proteins expressed. Because it was impossible to display all the proteins at once in the Bokeh application, we additionally describe an application of Holoviews/Datashader [HOLOVIEWS], [DATASHADER] to this data, which further highlights the extreme differences between oxygen rich surface waters and the oxygen poor mesopelagic. This application can be easily adapted to new datasets, and is already proving to be a useful tool for exploring patterns in ocean protein abundance.

Index Terms—oceanography, microbial ecology, biogeochemistry, omics, visualization, bokeh, datashader, holoviews, pandas, dask, jupyter

[§] Massachusetts Institute of Technology, Cambridge, MA
[‡] Woods Hole Oceanographic Institution, Woods Hole, MA
^{*} Corresponding author: msaito@whoi.edu

Copyright © 2018 Noelle A. Held et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Introduction

Oceanography is concerned with understanding the ocean as a holistic and dynamic system, integrating information from disciplines such as biology, chemistry, geology, and physics. But just how to incorporate this multivariate data is a key challenge in the field. For example, research expeditions commonly generate millions of data points, all with different formats, scales, and primary research goals. Scientific Python tools can help oceanographers synthesize multivariate information to make sense of trends; here we present an application to investigate metaproteome data from the oxygen poor central Pacific Ocean.

The tropical Pacific Ocean contains a naturally low-oxygen region called an oxygen minimum zone (OMZ) (Figure 1). Biological and chemical processes in the OMZ are different from surrounding oxygenated waters. For example, nitrification (use of ammonia or other organic nitrogen sources to fuel processes that typically use oxygen, in simplified form the reaction $\text{NH}_4 \rightarrow \text{NO}_2 \rightarrow \text{NO}_3$) is a key process in the OMZ but not present in oxygenated waters [ULLOA2012]. OMZs may represent a living proxy of the past anoxic ocean. They are also a picture into the future. Climate change driven by anthropogenic carbon dioxide emissions is causing ocean waters to be warmer and more stratified. This leads to deoxygenation processes and predicted expansion of OMZs [WRIGHT2012]. Thus, understanding the biogeochemistry of existing, natural OMZs is important for predicting conditions in the future ocean.

We travelled to the oxygen poor Pacific ocean in winter 2016 to study biological and chemical processes on the ProteOMZ research cruise (<https://schmidtocean.org/cruise/investigating-life-without-oxygen-in-the-tropical-pacific/>). To explore the biogeochemical processes in this region, we collected over 103 metaproteomics samples at various locations and depths, representing 56,577 identified proteins and over 6 million individual data points. In addition, we collected over 6,600 corresponding chemical and physical metadata points (18 variables) which provide context to the biological protein data. Proteins are the molecular machines driving biogeochemical transformations within microbial cells; as such, protein datasets provide a rich look at ecosystem function. To our knowledge this is the largest marine metaproteomics dataset to date.

In this paper, we describe our efforts to create an integrated proteomics and metadata visualization tool. The application is

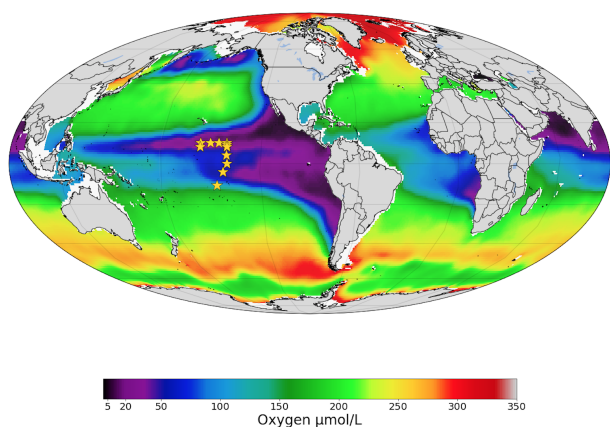


Fig. 1: Oxygen concentrations in the world ocean at 300m depth. Warm colors indicate more oxygen, cool colors indicate less. The sampling locations of the ProteOMZ cruise are overlaid as yellow stars. ProteOMZ samples the oxygen-depleted tropical Pacific region. Oxygen data: World Ocean Atlas [GARCIA2014]

intended as an exploratory tool for user-driven discovery of patterns in oceanographic protein abundance in relationship to hydrographic and ecological context. Using Bokeh [BOKEH] as the main visualization library, we developed an application that integrates multivariate data into interactive plots and tables. We begin by describing the data model, which emerged both from the inherent properties of the data and the constraints of the Bokeh library. We then describe an example in which we demonstrate major phylogenetic and functional differences between oxygen rich surface waters to oxygen poor mesopelagic waters. Due to performance constraints, the Bokeh application can only display a subset of the data. Therefore we additionally describe an application of Datashader implemented in Holoviews and Jupyter Notebook to visualize patterns in the entire dataset. This notebook further demonstrates functional partitioning between oxygen rich and poor waters, emphasizing the extremity of these biogeochemical differences. We conclude with a brief discussion of the benefits and drawbacks of our data construction and library choices, as well as some recommendations for developers and scientists working with these libraries.

Methods and Results

In situ sampling and data acquisition

Samples were collected in January-February 2016 at 14 locations (stations) in the tropical Pacific ocean. At each station, large volume in situ pumps were deployed at multiple depths in the water column. For each pump, hundreds of liters of water were passed through stacked 51 μM , 3 μM and 0.2 μM filters. The data described here is for the 0.2-3 μM filter range which includes most single cell phytoplankton and free living heterotrophic bacteria. More detail on proteomics analyses can be found in [SAITO2014]. The full sample collection and analysis methods for this dataset in particular will be reported in an upcoming publication.

Visualizing Hydrographic Data

We developed a visualization platform to explore the hydrographic data, which includes physical parameters such as temperature

ProteOMZ EXPEDITION 2016

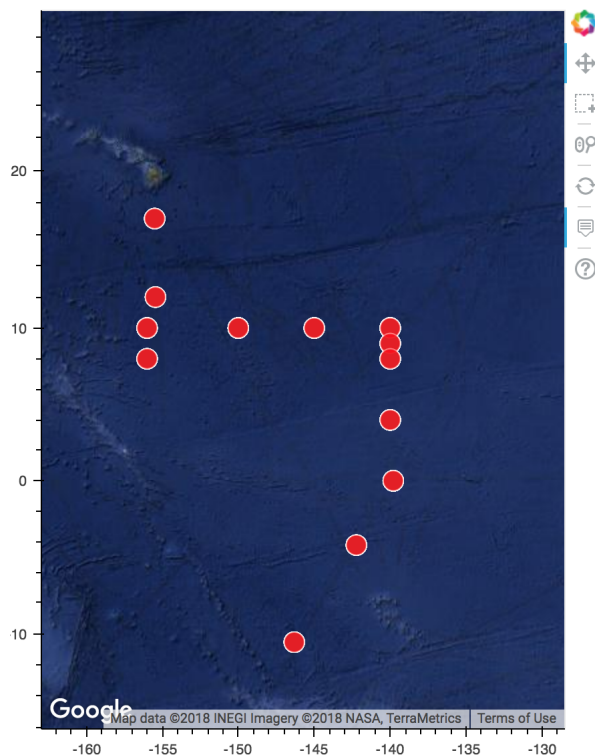


Fig. 2: Station map for the hydrographic data.

and chemical parameters such as ammonium concentrations. The visualization was written with Bokeh in the the Jupyter Notebook interface and produces a standalone html document as the output. This allows the document to be shared with colleagues and, importantly, does not require them to have bokeh or even python installed on their machine. The visualization consists of a map rendered in Google Maps using the `gmap` function in Bokeh (Figure 2a) and scatter plots showing the vertical distribution of the hydrographic parameters throughout the water column, with surface values at the top (Figure 2b). The plots are arranged with `gridplot`. This visualization is fed from a hydrographic data CSV file, where the data for each variable is in a separate column. This facilitates ingestion into Bokeh's `ColumnDataSource`, allowing the plots to be linked. Thus, when the user selects data from one plot, corresponding data for that location is highlighted in the other plots.

Bokeh Application

The main product of this work is a fully interactive Bokeh server application, which integrates protein quantitative data, protein annotations, and hydrographic data. For full interactivity among plots, Bokeh requires data to be in a single 2D `ColumnDataSource`. Thus, the first challenge we faced was how to compress our multidimensional data into a 2D format that could be accessed by multiple plots and updated via widgets. The protein quantitative data is a CSV formatted output which is generated directly from the common proteomics analysis program Scaffold [SCAFFOLD]. For illustrative purposes in this paper we use a truncated CSV file containing 15,000 of the nearly 60,000 identified proteins. However, we have had success using the entire 60,000 protein dataset.

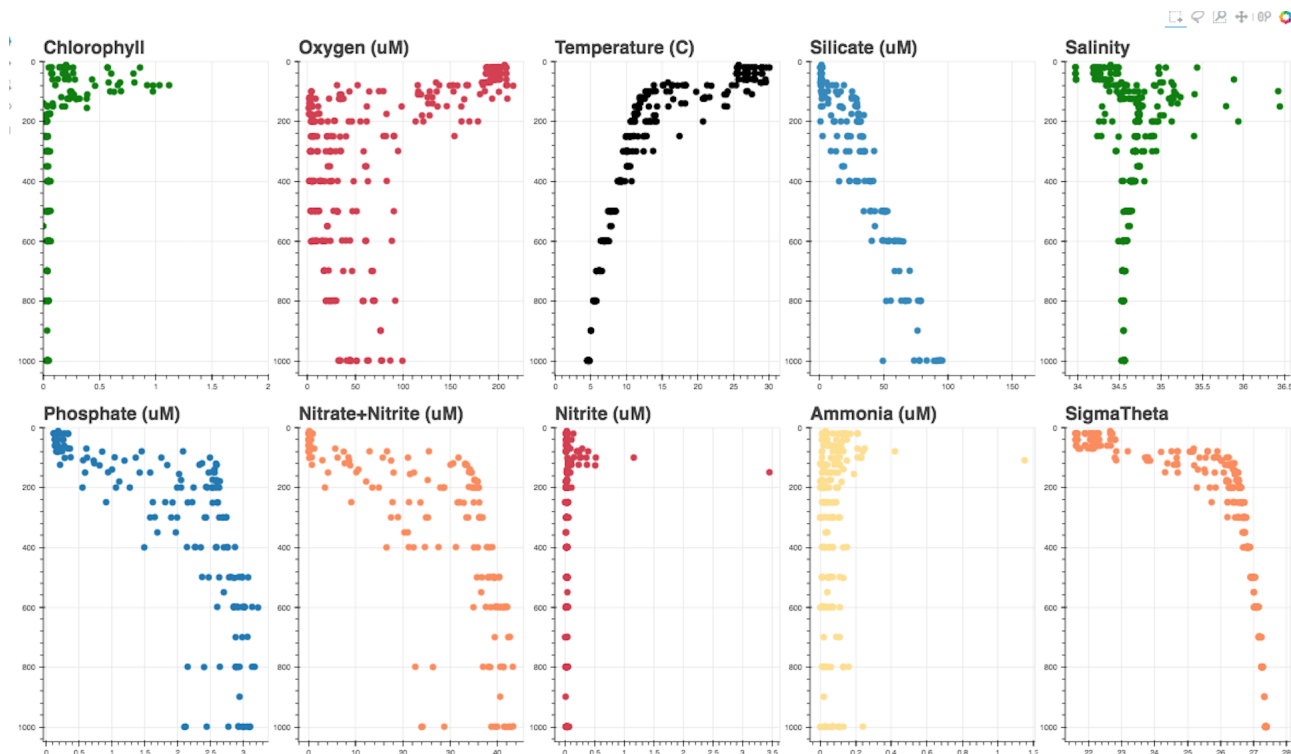


Fig. 3: Hydrographic data as a function of water column depth. The file is exported by Bokeh as a standalone html document, allowing it to be easily shared with collaborators.

The CSV file is read as a pandas dataframe [PANDAS] and consists of 103 rows (one per each unique sampling location and depth) and over 15,000 columns, where each column represents a different protein that was identified in the field sample. This 15,000 protein dataset is a subset of the full protein dataset of 60,000 proteins. The protein annotation information is read as a separate file and includes taxonomic and functional information about each protein in the dataset. Finally, the hydrographic data consists of 103 rows, again, one per each unique sampling location and depth) and 16 columns each containing a hydrographic or chemical parameter also measured on the expedition. We combined all three of these dataframes into a combined data model, allowing the entire application to be fed from ColumnDataSources generated from slices of a single Pandas dataframe (Figure 4). This facilitates connectivity among the plots via tools such as hover and tap, and allows the user to explore all the visualizations using widgets for protein annotation and hydrographic data.

We now describe a use case to demonstrate the utility of the application (Figure 5-8). On initial load, the user can see a map of the ProteOMZ 2016 sampling locations (Figure 5). The user can select a Station via a widget and display a vertical distribution of all of the proteins identified at this station throughout the water column, from surface to deep. Hovering over a protein in the vertical distribution profile displays its identity. The vertical distribution, protein annotation table, and protein vs. hydrographic data charts are directly linked since they are fed through the same ColumnDataSource. Selecting a protein via the TapTool highlights it in the vertical profile, protein annotation table, and in the Protein vs. Hydrographic data chart. A user who is interested in a specific protein can select it from the table, which updates the vertical line profile to highlight that protein. For instance, we can select the most abundant protein in the dataset at Station 5

and see that it is a nitrate oxidoreductase protein (Figure 6). The protein vs. hydrographic data chart displays protein abundance as a function of various hydrographic features, which can be selected by a widget. With the hydrographic widget we select nitrate (NO_3), a product of nitrification, and see that abundance of nitrate oxidoreductase is positively correlated with nitrate (Figure 7). The protein is negatively correlated with its reactant ammonium (NH_4), and also with the intermediary product nitrite (NO_2). Consistent with the idea that nitrification is prevalent in oxygen minimum zones, we see that the protein is negatively correlated with oxygen (O_2) concentrations.

Selecting a station additionally populates a vertical profile of the total number of unique proteins identified (line) and number of peptide-to-spectrum matches expressed on a log scale (bubble) at each depth sampled. In proteomics, we do not measure proteins but instead parts of proteins called peptides, which are then matched to spectra that are predicted in silico from a genome database. The peptide-to-spectrum match indicates the total number of peptides identified (non unique). Typically the number of peptide-to-spectrum matches is related to the number of unique peptides identified; we see this reflected in the data at Station 5. For instance, we see that at depths 200m and below there are more proteins and more peptide-to-spectrum matches than in surface waters. However, though the number of unique proteins is approximately constant between 200 and 500m, the number of PSMs varies.

So far we have looked only at protein function, but a user may also be interested in taxonomic origin of the proteins. At Station 5, we see in the Diversity of Microbial Proteins bar graph that most of the proteins we identified are from the group “Other Bacteria,” which encompasses most heterotrophic bacteria including the nitrifying bacteria (Figure 8). There are also many

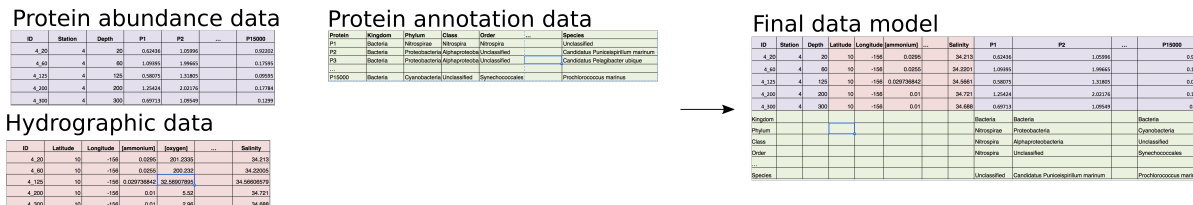


Fig. 4: Data model for integrating protein abundance, protein annotation, and hydrographic data into a single Bokeh ColumnDataSource, allowing for interactivity among the visualizations in the application.

Prochlorococcus and Pelagibacter proteins in the dataset, which is consistent with the fact that these cells are among the most abundant in the ocean [EGGLESTON2016]. A user can select a specific taxon with the taxon widget; for example, we can select “Prochlorococcus” from the taxon widget and redisplay the data (Figure 5). We can now see that Prochlorococcus, a photosynthetic cyanobacterium, is present primarily in the sunlit surface waters above 120m. If we display “Other Bacteria,” we can see that indeed that the heterotrophic nitrifying bacteria are highly abundant in the oxygen-depleted waters beginning around 200m. Thus with just a few clicks we can explore major taxonomic and functional regimes throughout the oxic and suboxic water column.

Application of Datashader

We quickly discovered that attempting to display over 15,000 lines on a single Bokeh plot was infeasible. We thus display only the top 5% most abundant proteins but allow the user to adjust this percentage via the Percentile slider. When the application is run via Bokeh server on a single laptop, only the top 5-10% of proteins can be displayed without significantly slowing down the visualizations. This alone is powerful - over 1000 proteins are displayed on the initial load, and the widgets allow the user to hone in on taxa and processes of interest such that meaningful information is still easy to find. However, it is clear that the data is oversampled and that proteins that are especially low abundance such as cell signalling and regulatory proteins are systematically “lost” in this visualization.

We used Datashader implemented in Holoviews and a Jupyter notebook to view the dataset in its entirety to see if major patterns in protein abundance emerge when all 15,000 test dataset lines are displayed. To improve performance in Datashader line, we re-formatted the dataframe to be two columns (x and y values) with each protein/depth set separated by NaNs. The dataframe was converted to a Dask dataframe [DASK] for performance reasons. Though this data model requires us to copy the “Depth” data 15,000 times, the performance improvement in the Datashader aggregation steps make this step worthwhile.

One question we can ask of the data is whether patterns emerge among proteins that are more or less abundant than average. We normalized the protein quantitation data by dividing each column by its average, such that the resulting data represents the fold-change in the protein in relationship to its mean over the entire water column. In the visualization, a value of 1 on the x axis suggests that protein abundance is equal to the mean; below 1 the protein is less abundant than average and above 1 the protein is more abundant.

In the datashader plot, the data is overlaid on itself such that areas with more saturated color indicates a high number of proteins with similar fold-change in concentration (Figure 8). This shows the partitioning of microbial proteins on depth. Proteins

that are abundant in the surface converge to 0, or “disapper” around 120m. At Station 5, the warm sunlight euphotic mixed layer ends at approximately 120m. These surface proteins are most likely attributed to Prochlorococcus, an abundant bacterium that lives only in sunlight waters. Below 120m, proteins attributed to heterotrophic bacteria become abundant.

Discussion

We designed a data integration and discovery tool for the ProteOMZ research expedition. In just a few clicks, the application allows users to explore trends in protein abundance and probe relationships between protein abundance and hydrographic data, and dial in to biological processes of interest. As an example we describe how we were able to rapidly investigate the taxonomic and functional differences between oxygen replete surface waters and the oxygen minimum mesopelagic. Since the application uses data from a common proteomics data file format, it will be simple to plug new oceanographic datasets into this application as they become available.

A key challenge to this project was building a data model that worked most efficiently with the libraries we selected. For instance, the Bokeh ColumnDataSource imposed a 2D structure on our multi-dimensional data. In Datashader we faced a similar issue, in which we discovered that aggregating 15,000 individual lines is prohibitively slow; by simply reformatting the data so the aggregation treats the data as individual points we could significantly improve performance. Learning about the constraints of these libraries was an important step in the process of creating this application, especially because we pushed the limits of the libraries. This required deep reading of user guides, API documentation, and Q/A repositories. We thus have two suggestions - 1) that scientists (and others) understand and carefully consider the data models and preferences of the libraries they plan to use before they begin the project and 2) that documentation of the data models and best practices in data formatting be more explicitly referenced in library user guides and be made easier to understand for the non-expert.

Another challenge we faced were problems with API stability. In large part this is due to the fact that we chose to work with libraries that are still in V0 release. We quickly learned to version control our code and used virtual environments to retain specific package versions. Luckily, since the projects are open source it is relatively easy to find information about recent changes, though this is not without frustration. For instance, the Bokeh application originally contained a donut chart, which has since been deprecated. We look forward to more stable releases of the Bokeh, Holoviews, and Datashader libraries, especially because we are now incorporating some of these visualizations into the upcoming Ocean Protein Portal (<http://proteinportal.who.edu/>), a

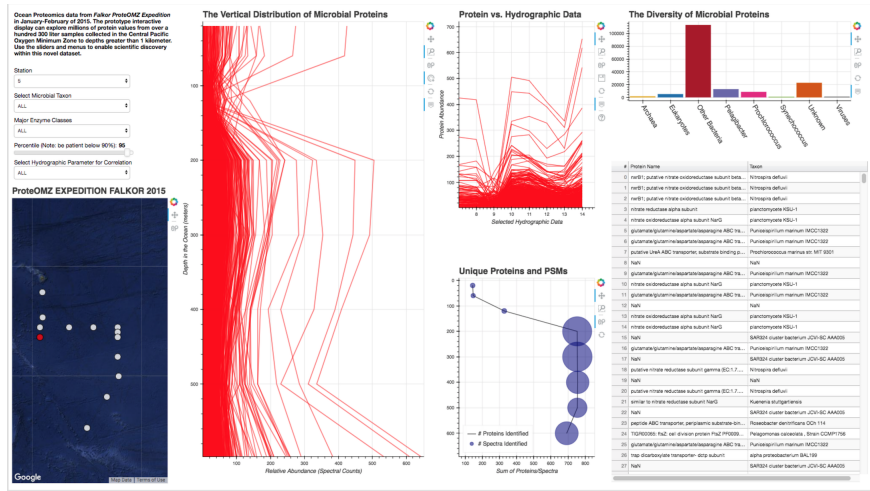


Fig. 5: Initial load of the Bokeh application.

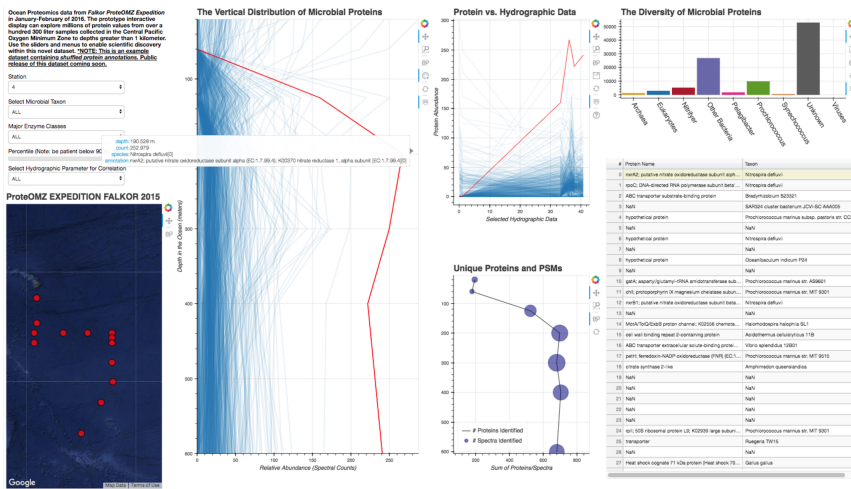


Fig. 6: Selecting on a single protein and investigating relationship to hydrographic data.

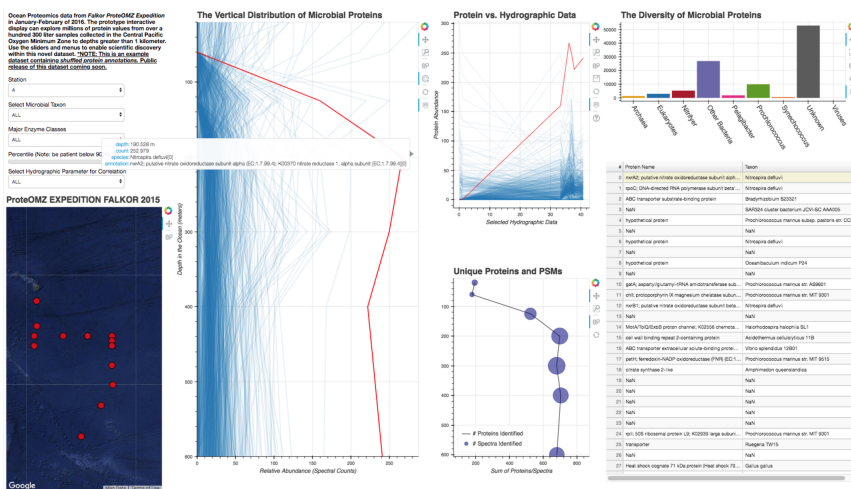


Fig. 7: Filtering on the taxon, we can see that Prochlorococcus proteins are present only in the upper 120m of the water column at this station.

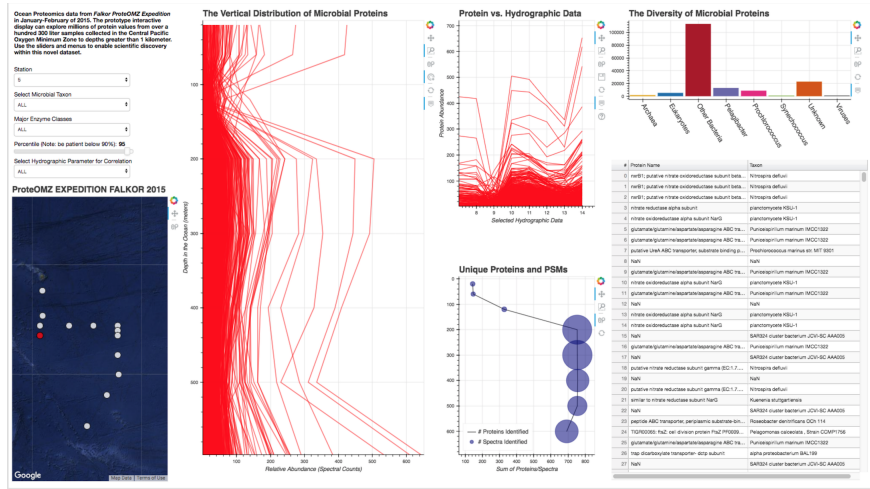


Fig. 8: Selecting “Other Bacteria,” we can see that the nitrifying bacteria become prevalent around 200m in the oxygen minimum zone.

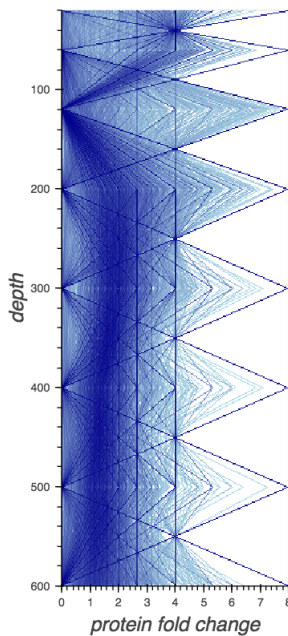


Fig. 9: Datashaded version of the vertical protein distribution plot, displaying all 15,000 proteins at Station 5. Each protein abundance is displayed as the difference from its average, so a value of >1 indicates a protein that is more abundant. A large number of Prochlorococcus proteins is present in the upper 120m; this collection of proteins disappears at the base of the euphotic zone. A large number of proteins is present in approximately the same fold change abundance throughout the mesopelagic region.

data sharing and discovery interface for marine metaproteomics data.

The main benefit of building these visualizations using Scientific Python tools is that scientists who are not primarily programmers can easily manipulate and maintain the code. The code is relatively straightforward, largely due to the fact that the Bokeh and in particular Holoviews backends do much of the heavy lifting. This makes it easier for colleagues to adapt the code to their own datasets. The linked charts in the Bokeh application allow for intuitive (read: more efficient) exploration of the data. In addition, charts generated by Bokeh, Datashader and Holoviews

are beautiful “out of the box.” This is an advantage when we share these visualizations not only with other scientific experts, but also with the general public during outreach events.

The visualizations we built are already proving to be useful. We discuss above just one high level example in which the application helps us to explore taxonomic and functional differences between oxic and suboxic water masses. Finer level analyses are sure to uncover even more exciting trends. We are already plugging in new datasets to the application. As mentioned above, many of these visualizations (in addition to some new ones, such as Holoviews Sankey plot) are being incorporated into the upcoming Ocean Protein Portal, which will make them even more accessible to the scientific community.

Code

Hydrography Visualization: https://github.com/maksaito/proteOMZ_hydrography_visualization Bokeh Application: https://github.com/maksaito/proteOMZ_visualization_app_public Datashader notebook: https://github.com/naheld/15000lines_datashader

Acknowledgements

This work is supported by a National Science Foundation Graduate Research Fellowship under grant number 1122274 (N. Held) and a NASA Postdoctoral Program Fellowship (J. Saunders). It is also supported by the Gordon and Betty Moore Foundation grant number 3782 (M. Saito) and National Science Foundation grant EarthCube 1639714.

REFERENCES

[BOKEH] Bokeh Project. <http://bokeh.pydata.org/>.
 [DASK] Dask Project. <https://dask.pydata.org/en/latest/>.
 [DATASHADER] Datashader Project. <http://datashader.org/index.html>.
 [EGGLESTON2016] Eggleston, E. M., & Hewson, I. (2016). Abundance of two Pelagibacter ubiquie bacteriophage genotypes along a latitudinal transect in the north and south Atlantic Oceans. *Frontiers in Microbiology*, 7(SEP), 1–9. <https://doi.org/10.3389/fmicb.2016.01534>
 [GARCIA2014] Garcia, H. E., R. A. Locarnini, T. P. Boyer, J. I. Antonov, O.K. Baranova, M.M. Zweng, J.R. Reagan, D.R. Johnson, 2014. World Ocean Atlas 2013, Volume 3: Dissolved Oxygen, Apparent Oxygen Utilization, and Oxygen Saturation. S. Levitus, Ed., A. Mishonov Technical Ed.; NOAA Atlas NESDIS 75, 27 pp.

- [HOLOVIEWS] Holoviews Project. <http://holoviews.org/>.
- [PANDAS] Pandas Project. <https://pandas.pydata.org/>.
- [SAITO2014] Saito, M. A., McIlvin, M. R., Moran, D. M., Goepfert, T. J., DiTullio, G. R., Post, A. F., & Lamborg, C. H. (2014). Multiple nutrient stresses at intersecting Pacific Ocean biomes detected by protein biomarkers. *Science* (New York, N.Y.), 345(6201), 1173–7. <https://doi.org/10.1126/science.1256450>
- [SCAFFOLD] Scaffold, Proteome Software <http://www.proteomesoftware.com/products/scaffold/>
- [ULLOA2012] Ulloa, O., Canfield, D. E., DeLong, E. F., Letelier, R. M., & Stewart, F. J. (2012). Microbial oceanography of anoxic oxygen minimum zones. *Proceedings of the National Academy of Sciences*, 109(40), 15996–16003. <https://doi.org/10.1073/pnas.1205009109>
- [WRIGHT2012] Wright, J. J., Konwar, K. M., & Hallam, S. J. (2012). Microbial ecology of expanding oxygen minimum zones. *Nature Reviews Microbiology*, 10(6), 381–394. <https://doi.org/10.1038/nrmicro2778>

Binder 2.0 - Reproducible, interactive, sharable environments for science at scale

Project Jupyter^{‡‡}, Matthias Bussonnier^{||†}, Jessica Forde^{‡‡}, Jeremy Freeman^{‡‡‡}, Brian Granger^{§†}, Tim Head^{¶†}, Chris Holdgraf^{||*}, Kyle Kelley^{†††}, Gladys Navarte^{**†}, Andrew Osheroﬀ^{‡‡‡}, M Pacer^{†††}, Yuvi Panda^{||†}, Fernando Perez^{||†}, Benjamin Ragan-Kelley^{**†}, Carol Willing^{§†}

<https://youtu.be/KcC0W5LP9GM>



Abstract—Binder is an open source web service that lets users create sharable, interactive, reproducible environments in the cloud. It is powered by other core projects in the open source ecosystem, including JupyterHub and Kubernetes for managing cloud resources. Binder works with pre-existing workflows in the analytics community, aiming to create interactive versions of repositories that exist on sites like GitHub with minimal extra effort needed. This paper details several of the design decisions and goals that went into the development of the current generation of Binder.

Index Terms—cloud computing, reproducibility, binder, mybinder.org, shared computing, accessibility, kubernetes, dev ops, jupyter, jupyterhub, jupyter notebooks, github, publishing, interactivity

Binder is a free, open source, and massively publicly available tool for easily creating sharable, interactive, reproducible environments in the cloud.

The scientific community is increasingly unified around reproducibility. A survey in 2016 of 1,576 researchers reported that 90% of respondents believed there exists a reproducibility crisis in the scientific community. A majority of respondents also reported difficulty reproducing the work of colleagues [Bak16]. Similar results have been reported in the cell biology community [The] and the machine learning community [Pin17]. Making research reproducible requires pursuing two sub-goals, both of which are difficult to achieve:

- **technical reproducibility:**
making reproducible scientific results possible at all
- **practical reproducibility:**
enabling others to reproduce results without difficulty

Both technical and practical reproducibility depend upon the software and technology available to researchers at any moment in time. With the growth in open source tools for data analysis,

as well as the “data heavy” approach many fields are adopting, these problems become more complex yet more tractable than ever before.

Fortunately, as the problem has grown more complex, the open source community has risen to meet the challenge. Tools for packaging analytics environments into “containers” allow others to re-create the computational environments needed to run analyses and evaluate results. Online communities make it easier to share and discover scientific results. A myriad of open source tools are freely available for doing analytics in open and transparent ways. New paradigms for writing code and displaying results in rich, engaging formats allow results to live next to the prose that explains their purpose.

However, manual implementation of this processes is complex, and reproducing the full stack of another person’s work is too labor intensive and error-prone for day-to-day use. A recent study of scientific repositories found that citation of “both visualization tools as well as common software packages (such as MATLAB) was a widespread failure” [SSM18]. As a result, the technical barriers limit practical reproducibility. To lower the technical barriers of sharing computational work, we introduce Binder 2.0, a tool that we believe makes reproducibility more practically possible.

An overview of Binder

Binder consists of a set of tools for creating sharable, interactive, and deterministic environments that run on personal computers and cloud resources. It manages the technical complexity around:

- creating containers to capture a code repository and its technical environment;
- generating user sessions that run the environment defined in those containers; and
- providing links that users can share with others to allow them to interact with these environments.

Binder is built on modern-day tools from the open source community and is itself fully open source for others to use.

You can access a public deployment of Binder at mybinder.org, a web service that the Binder and JupyterHub teams run as a demonstration of the BinderHub technology and as digital public infrastructure for those who wish to share Binder links so that others may interact with their code repositories. It is meant to be a

† These authors contributed equally.

‡ Project Jupyter

|| UC Berkeley

‡‡

§ Cal Poly, San Luis Obispo

¶ Wild Tree Tech, Switzerland

* Corresponding author: choldgraf@berkeley.edu

†† Netflix

** Simula Research Lab

testing ground for different use cases in the Binder ecosystem as well as a public service for the scientific and educational community. mybinder.org serves nearly 9,000 daily sessions, and has already been used for reproducible publishing¹, sharing interactive course materials², at the university and high-school level, creating interactive package documentation in Python³ with Sphinx Gallery, and sharing interactive content that requires a language-specific kernel in order to run⁴.

Binder continues in the tradition of promoting "the complete software development environment and the complete set of instructions which generated the figures" [BD95] by effortlessly providing these tools to the general public in the cloud. The first iteration of Binder was released in 2016 [FO16] and provided a prototype that managed reproducible user environments in the cloud. In the years since, there have been several advances in technology for managing cloud resources, serving interactive user environments, and creating reproducible containers for analytics. Binder 2.0 utilizes these new tools, and it is more scalable and maintainable, is easier to deploy, and supports more analytic and scientific workflows than before. While previous work has specified methods or file formats for the sharing of research [BD95] [GL07] [LV15], Binder only requires configuration files typically seen in contemporary software development. Related online platforms for reproducibility also have specific front ends for presenting research and commands for running code [AESM17] [LV15] [SHP12], while Binder flexibly allows users to interact with a repository using modern data science tools such as RStudio, Jupyter Notebook, and JupyterLab. By containerizing the environment and using these front-end data science tools, Binder prioritizes an interactive user experience so that "someone else can discover it for themselves" [Som18].

At the highest level, Binder is a particular combination of open source tools to achieve the goal of sharable, reproducible environments. This paper lays out the technical vision of Binder 2.0, including the guiding principles and goals behind each piece of technology it uses. It also discusses the guiding principles behind the *new* open source technology that the project has created.

Guiding Principles of Binder

Several high-level project goals drive the development of Binder 2.0. These are outlined below:

Deployability. Binder is driven by open source technology, and the BinderHub server should be deployable by a diverse representation of people in the scientific, publishing, and data analytic communities. This often means that it must be maintained by people without an extensive background in cloud management and dev-ops skills. BinderHub (the underlying technology behind Binder) should thus be deployable on a number of cloud frameworks, and with minimal technical skills required.

Maintainability. Deploying a service on cloud resources is important but happens less frequently than *maintaining* those cloud resources all day, every day. Binder is designed to utilize modern-day tools in cloud orchestration and monitoring. These

tools minimize the time that individuals must spend ensuring that the service performs as expected. Recognizing the importance of maintainability, the Binder team continues to work hard to document effective organizational and technical processes around running a production BinderHub-powered service such as mybinder.org. The goal of the project is to allow a BinderHub service to be run without specialized knowledge or extensive training in cloud orchestration.

Pluggability. Binder's goal is to make it easier to adopt and interact with existing tools in the open source ecosystem. As such, Binder is designed to work with a number of open source packages, languages, and user interfaces. In this way, Binder acts as glue to bring together pieces of the open source community, and it easily plugs into new developments in this space.

Accessibility. Binder should be as accessible as possible to members of the open source, scientific, educational, and data science communities. By leveraging pre-existing workflows in these communities rather than requiring people to adopt new ones, Binder increases its adoption and user acceptance. Input and feedback from members of those communities guide future development of the technology. As a key goal, Binder should support pre-existing scientific workflows and improve them by adding sharability, reproducibility, and interactivity.

Usability. Finally, the Binder team wants simplicity and fast interaction to be core components of the service. Minimizing the number of steps towards making your work sharable via Binder helps provide an effective user experience. Consumers of shared work must be able to quickly begin using the Binder repository that another person has put together. To achieve these goals, creating multiple ways in which people can use Binder's services is key. For example, easily sharing a link to the full Binder interface and offering a public API endpoint to request and interact with a kernel backed by an arbitrary environment increase usability.

In the following sections, we describe the three major technical components that the Jupyter and Binder teams have developed for the Binder project—JupyterHub, repo2docker, and BinderHub. All are open source, and rely heavily on other tools in the open source ecosystem. We'll discuss how each feeds into the principles we've outlined above.

Scalable interactive user sessions

Binder runs as either a public or a private web service, and it needs to handle potentially large spikes in user sessions as well as sustained user activity over several minutes of time. It also needs to be deployable on a number of cloud providers in order to avoid locking in the technology to the offerings of a single cloud service. To accomplish this Binder uses a deployment of JupyterHub that runs on Kubernetes, both of which contribute to BinderHub's scalability and maintainability.

JupyterHub, an open source tool from the Jupyter community, provides a centralized resource that serves interactive user sessions. It allows definition of a computational environment (e.g. a Docker image) that runs the Jupyter notebook server. A core principle of the Jupyter project is to be language- and workflow-agnostic, and JupyterHub is no exception. JupyterHub can be used to run dozens of languages served with a variety of user interfaces, including Jupyter Notebooks [Bus18], JupyterLab [Pro17b], RStudio [Pro17a], Stencila [RN18], and OpenRefine [Hea18].

1. <https://github.com/minrk/ligo-binder>

2. <https://www.inferentialthinking.com/chapters/01/3/plotting-the-classics.html>

3. https://sphinx-gallery.readthedocs.io/en/latest/advanced_configuration.html#binder-links

4. <http://greenteapress.com/wp/think-dsp/>

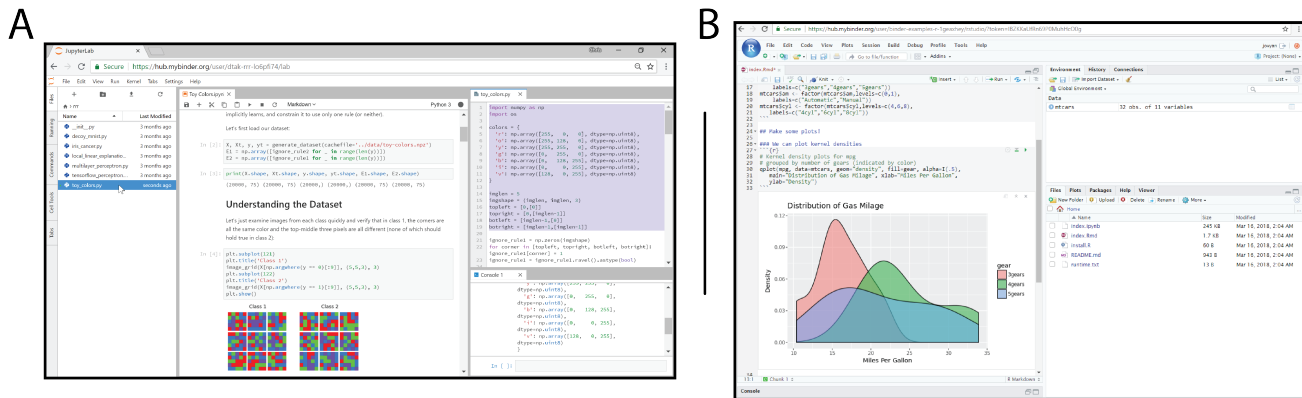


Fig. 1: Two example user interfaces that users can run within Binder. Because BinderHub uses a JupyterHub for hosting all user sessions, one can specify an environment that serves any Jupyter-supported user interface, provided that it can run via the browser. A. Examining image data from Ross et al. on Binder with JupyterLab [RHDV17]. JupyterLab provides access to the file system (left column), a notebook interface (middle column), as well as traditional script files and interactive kernels (right column). B. An RStudio interface running the modern RStudio and tidyverse stack. In both cases, users can explore the code and make their own modifications from within the Binder session, without any need to manually install dependencies.

Another key benefit of JupyterHub is that it is straightforward to run on Kubernetes, a modern-day open source platform for orchestrating computational resources in the cloud. Kubernetes can be deployed on most major cloud providers, self-hosted infrastructure (such as OpenStack deployments), or even on an individual laptop or workstation. For example, Google Cloud Platform, Microsoft Azure, and Amazon AWS each have managed Kubernetes clusters that run with minimal user intervention. Thus, it is straightforward to deploy JupyterHub on any major cloud provider.

Kubernetes is designed to be relatively self-healing, often automatically resolving problems that would normally disrupt the service. It also has a declarative syntax for defining the cloud resources that are needed to run a web service. Thus, maintainers can update a JupyterHub running on Kubernetes with minimal changes to configuration files for the deployment, providing the flexibility to configure the JupyterHub as needed, without requiring a lot of hands-on intervention and tinkering.

Finally, Kubernetes is both extremely scalable and battle-tested because it was originally developed to run Google’s web services. A cloud orchestration tool that can handle the usage patterns of a service like Gmail can almost certainly handle the analytics environments that are served with Binder. In addition, by using Kubernetes, Binder (with JupyterHub) leverages the power of Kubernetes’ strong open source community. As more companies, organizations, and universities adopt and contribute to the tool, the Binder community will benefit from these advances.

There are several use-cases of JupyterHub being used for shared, interactive computing. For example, UC Berkeley hosts a Foundations in Data Science [Ber] course that serves nearly 1,000 interactive student sessions simultaneously. The Wikimedia foundation also uses JupyterHub to facilitate users accessing the Wikipedia dataset [Wik], allowing them to run bots and automate the editing process with a Jupyter interface. Finally, organizations such as the Open Humans Project provide a JupyterHub for their community [Ope] to analyze, explore, and discover interesting patterns in a shared dataset.

Deterministic environment building - Repo2Docker

Docker [Doc] is extremely flexible, and has been used throughout the scientific and data science community for standardizing environments that are sharable with other people. A Docker image contains nearly all of the pieces necessary to re-run an analysis. This provides the right balance between flexibility (e.g. a Docker image can contain basically any environment) and being lightweight to deploy and store in the cloud. JupyterHub can serve an arbitrary environment to users based off of a Docker image, but how is this image created in the first place?

While it is possible (and common) to hand-craft a Docker image using a set of instructions called a Dockerfile, this step requires a considerable amount of knowledge about the Docker platform, making it a high barrier to the large majority of scientists and data analysts. Binder’s goal is to operate with many different workflows in data analytics, and requiring the use of a Dockerfile to define an environment is too restrictive.

At the same time, the analytics community already makes heavy use of online code repositories, often hosted on websites such as GitHub [Git] or Bitbucket [Atl]. These sites are home to tens of thousands of repositories containing the computational work for research, education, development, and general communication. Best practices in development already dictate storing the requirements needed (in text files such as `environment.yml`) along with the code itself (which often lives in document structures such as Jupyter Notebooks or RMarkdown files). As a result, in many cases the repository already contains all the information needed to build the required environment.

Binder’s solution to this is a lightweight tool called “repo2docker” [Pro17c]. It is an open source command line tool that converts code repositories into a Docker image suitable for running with JupyterHub. Repo2docker:

- 1) is called with a single argument, a path to a git repository, and optionally a reference to a git branch, tag, or commit hash. The repository can either be online (such as on GitHub or GitLab) or local to the person’s computer.
- 2) clones the repository, then checks out the reference that it has been passed (or defaults to “master”).

- 3) looks for one or more “configuration” files that are used to define the environment needed to run the code inside the repository. These are generally files that *already exist* in the data science community. For example, if it finds a `requirements.txt` file, it assumes that the user wants a Python installation and installs everything inside the file. If it finds an `install.R` file, it assumes the user wants RStudio available, and pre-installs all the packages listed inside.
- 4) constructs a `Dockerfile` that builds the environment specified by the configuration files, and that is meant to be run via a Jupyter notebook server.
- 5) builds an image from this `Dockerfile`, and then registers it online with a Docker repository of choice.

Repo2docker aims to be flexible in the analytics workflows it supports, and it minimizes the amount of effort needed to support a *new* workflow. A core building block of `repo2docker` is the “Build Pack” - a class that defines all of the operations needed to construct the environment needed for a particular analytics workflow. These Build Packs have a `detect` method that returns `True` when a particular configuration file is present (e.g. `requirements.txt` will trigger the Python build pack). They also have a method called `get_assemble_scripts` that inserts the necessary lines into a `Dockerfile` to support this workflow.

For example, below we show a simplified version of the Python build pack in `repo2docker`. In this case, the `detect` method looks for a `requirements.txt` file and, if it exists, triggers the `get_assemble_scripts` method, which inserts lines into the `Dockerfile` that install Python and `pip`. Binder uses `repo2docker` to build repository images dynamically.

```
class PythonBuildPack(CondaBuildPack):
    """Setup Python for use with a repository."""

    def __init__(self):
        ...

    def get_assemble_scripts(self):
        """Return build-steps specific to this repo."""
        assemble_scripts = super().get_assemble_scripts()
        # KERNEL_PYTHON_PREFIX is the env with the kernel
        # whether it's distinct from the notebook
        # or the same.
        pip = '${KERNEL_PYTHON_PREFIX}/bin/pip'

        # install requirements.txt in the kernel env
        requirements_file = self.binder_path(
            'requirements.txt')
        if os.path.exists(requirements_file):
            assemble_scripts.append((
                '${NB_USER}',
                '{} install --no-cache-dir -r {}'.format(
                    pip, requirements_file)
            ))
        return assemble_scripts

    def detect(self):
        """Check if repo builds w/ Python buildpack."""
        requirements_txt = self.binder_path(
            'requirements.txt')
        return os.path.exists(requirements_txt)
```

Repo2docker also supports more generic configuration files that are applied regardless of the particular Build Pack that is detected. For example, a file called “postBuild” will be run from the shell after all dependencies are installed. This is often used to pre-compile code or download datasets from the web.



Turn a GitHub repo into a collection of interactive notebooks

Have a repository full of Jupyter notebooks? With Binder, open those notebooks in an executable environment, making your code immediately reproducible by anyone, anywhere.

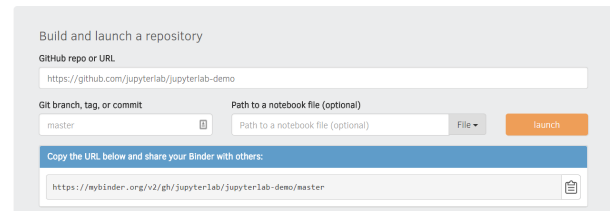


Fig. 2: The BinderHub user interface. Users input a link to a public git repository. Binder will check out this repository and build the environment needed to run the code inside. It then provides you a link that can be shared with others so that they may run an interactive session that runs the repository’s code.

Finally, in the event that a particular setup is not natively supported, `repo2docker` will also build a Docker image from a plain `Dockerfile`. This means users are never blocked by the design of `repo2docker`.

By modularizing the environment generation process in this fashion, it is possible to mix and match environments that are present in the final image. `Repo2docker`’s goal is to allow for a fully composable analytics environment. If a researcher requires Python 2, 3, RStudio, and Julia, simultaneously for their work, `repo2docker` should enable this.

In addition, by capturing pre-existing workflows rather than requiring data analysts to adopt new ones, there is a minimal energy barrier towards using `repo2docker` to deterministically build images that run a code repository. For example, if the following `requirements.txt` file is present in a repository, `repo2docker` will build an image with Python 3 and the packages `pip` installed.

```
$ cat requirements.txt
numpy
scipy
matplotlib
```

While the following file name/content will install RStudio with these R commands run before building the Docker image.:

```
$ cat binder/install.R
install.packages("ggplot2")
```

```
$ cat binder/runtime.txt
r-2017-10-24
```

In this case, the date specified in `runtime.txt` instructs `repo2docker` to use a specific MRAN repository [Mic] date. In addition, note that these files exist in a folder called `binder/` (relative to the repository root). If `repo2docker` discovers a folder of this name, it will build the environment from the contents of this folder, ignoring any configuration files that are present in the project’s root. This allows users to dissociate the configuration files used to build the package from those used to share a Binder link.

By facilitating the process by which researchers create these reproducible images, `repo2docker` addresses the “works for me” problem that is common when sharing code. There are no longer

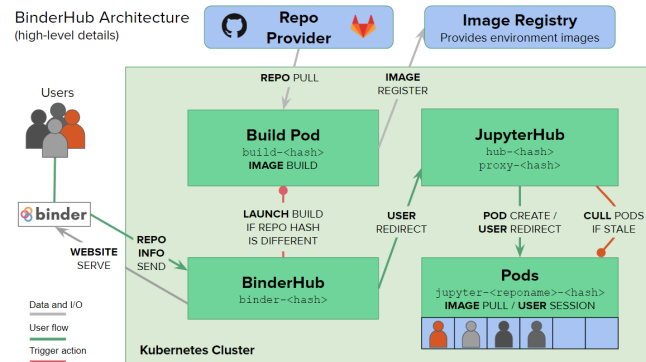


Fig. 3: The BinderHub architecture for interactive GUI sessions. Users connect to the Binder UI via a public URL. All computational infrastructure is managed with a Kubernetes deployment (light green) managing several pods (dark green) that make up the BinderHub service. Interactive user pods (blue squares) are spawned and managed by a JupyterHub.

breaking differences in the environment of two users if they are running code from the same image generated by repo2docker. Additionally, researchers can use repo2docker to confirm that all of the information needed to recreate their analysis is contained within their configuration files, creating a way to intuitively define “recipes” for reproducing one’s work.

A web-interface to user-defined kernels and interactive sessions - BinderHub

JupyterHub can serve multiple interactive user sessions from pre-defined Docker images in the cloud. Repo2docker generates Docker images from the files in a git repository. BinderHub is the glue that binds these two open source tools together. It uses the building functionality of repo2docker, the kernel and user-session hosting of JupyterHub, and a Docker registry that connects these two processes together. BinderHub defines two primary patterns of interaction with this process: sharable, interactive, GUI-based sessions; and a REST API for building, requesting, and interacting with user-defined kernels.

The BinderHub User Interface

The primary pattern of interaction with BinderHub for an author is via its “build form” user interface. This form lets users point BinderHub to a public git repository. When the form is filled in and the “launch” button is clicked, BinderHub takes the following actions:

- 1) **Check out the repository** at the version that is specified.
- 2) **Check the latest commit hash.** BinderHub compares the version specified in the URL with the versions that have been previously built for this repository in the registry (if a branch is given, BinderHub checks the latest commit hash on this branch).
- 3) If the version has *not* been built, **launch a repo2docker process** that builds and registers an image from the repository, then returns a reference to the registered image.
- 4) **Create a temporary JupyterHub user account** for the visitor, with a private token.
- 5) **Launch a JupyterHub user session** that sources the repo2docker image in the registry. This session will serve the environment needed to run the repository, along with any GUI that the user specifies.
- 6) **Clean up the user session.** Once the user departs, Binder destroys the temporary user ID for the user’s

unique session, as well as their temporary files from their interactive session (steps 4 and 5). The Docker image for the repository persists, and will be used in subsequent launch attempts (as long as the repository commit hash does not change).

Once a repository has been built with BinderHub, authors can then share a URL that triggers this process. URLs for BinderHub take the following form:

```
<bhub-url>/v2/<repoprovider>/<org>/<reponame>/<ref>
```

For example, the URL for the binder-examples repository that builds a Julia environment is

```
mybinder.org/v2/gh/binder-examples/julia-python/master
```

When a user clicks on this link, they will be taken to a brief loading page as a user session that serves this repository is created. Once this process is finished, they can immediately start interacting with the environment that the author has created.

The BinderHub REST API

While GUIs are preferable for most human interaction with a BinderHub, there are also situations when a programmatic or text-based interaction is preferable. For example, someone may wish to use BinderHub to request arbitrary kernels that power computations underlying a completely different GUI. For these use cases, BinderHub also provides a REST API that controls all of the steps described above.

BinderHub currently provides a single REST endpoint that allows users to programmatically build and launch Binder repositories. It takes the following form:

```
<bhub-url>/build/<provider>/<spec>
```

This follows a similar pattern to BinderHub’s sharable URLs. For example, the following API request results in a Binder environment for the JupyterLab example repository on mybinder.org:

```
mybinder.org/build/gh/binder-examples/jupyterlab/master
```

Accessing this endpoint will trigger the following events:

- 1) Check if the image for this URL exists in the BinderHub cached image registry. If yes, launch it.
- 2) If it doesn’t exist in the image registry, check if a build is currently running. If there is **not**, then start a build process. If there **is**, then attach to the pre-existing build process.

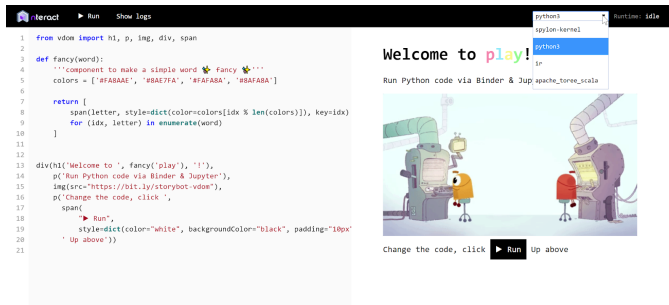


Fig. 4: *play.nteract.io* [nte16] is a GUI front-end that connects to the *mybinder.org* REST API. When a user opens the page, it requests a kernel from *mybinder.org* according to the environment chosen in the top-right menu. Once *mybinder.org* responds that it is ready, users can execute code that will be sent to their Binder kernel, with results displayed on the right.

- 3) Stream logs from the build process to the user.
- 4) If the build succeeds, contact the JupyterHub API, telling it to launch a user server with the environment that has just been built.
- 5) Once the server is launched, display a message showing the URL where they can connect to the notebook server (and thus connect with the Jupyter Notebook Server REST API).

Information about the process above is streamed to the user via a persistent HTTP connection with structured JSON messages via the EventStream protocol. Here's an example of the output for the above build:

```
data: {"phase": "built",
      "imageName": "gcr.io/binder-prod/r2d-051...",
      "message": "Found built image, launching..."}
data: {"phase": "launching", "message": "Launching..."}
data: {"phase": "ready",
      "message": "server running at <POD-URL>",
      "url": "<POD-URL>",
      "token": "<POD-TOKEN>"}
```

In this case, the user can then access the value in `url`: to use their Binder session (either via their browser, or programmatically via the notebook server REST API served at this URL).

There are already several examples of services that use BinderHub's REST API to run webpages and applications that utilize arbitrary kernel execution. For example, thebelab [Min] makes it possible to deploy HTML with code blocks that are powered by a BinderHub kernel. The website creator can define the environment needed to run code on the page, and the end user can generate interactive code output once they visit the webpage. There are also several applications that use BinderHub's kernel API to power their computation. For example, the nteract [nte16] project uses BinderHub to run an interactive code sandbox that serves an nteract interface and can be powered by arbitrary kernels served by BinderHub.

BinderHub is permissively licensed and intentionally modular in order to serve as many use cases as possible. Our goal is to provide the tools to allow any person or organization to provide arbitrary, user-defined kernels that run in the cloud. The Binder team runs one such service as a proof-of-concept of the technology, as well as digital public infrastructure that can be used

to share interactive code repositories. This service runs at the URL mybinder.org and will be discussed in the final section.

Mybinder.org: Maintaining and sustaining a public service

In addition to providing a showcase for the technical components of the BinderHub, repo2docker, and JupyterHub architecture, the Binder project is also a case study in the maintenance and deployment of an open-source service. Managing and providing a site such as mybinder.org is not trivial, with challenges in team operations, maintaining service stability without any full-time staff, and exploring models for keeping the project financially sustainable over time. This final section describes recent efforts to address some of these questions, and to explore possible outcomes for others.

The Binder team (and thus mybinder.org) runs on a model of transparency and openness in the tools it creates as well as the operations of mybinder.org. The Binder team has put together several group processes and documentation to facilitate maintaining this public service, and to provide a set of resources for others who wish to do the same. For example, the Binder Site Reliability Guide⁵ is continuously updated with team knowledge, incident reports, helper scripts, and a description of the technical deployment at mybinder.org. There are also several data streams that the Binder team routinely makes available for others who are interested in deploying and maintaining a BinderHub service. For example, the Binder Billing⁶ repository shows all of the cloud hardware costs for the last several months of mybinder.org operation. In addition, the Binder Grafana board⁷ shows a high-level view of the status of the BinderHub, JupyterHub, and Kubernetes processes underlying the service.

Cost of running the public Binder service

The Binder team has designed the public service to be as cost effective as possible. mybinder.org restricts users to one CPU and two GB of RAM. We save a great deal by not providing users with persistent storage across sessions. Users can only access public git repositories and are restricted in the kinds of network I/O that can take place. In addition, a BinderHub deployment efficiently uses its resources in order to avoid over-provisioning cloud resources.

The decision to avoid the notion of a user "identity" in particular has strong effects on the cost of running a BinderHub server. Because users do not require persistent storage (e.g. the content of any changes they make to Jupyter Notebooks throughout a session), a significant cost of running a JupyterHub is avoided. In addition, a BinderHub deployment can efficiently use the resources available to it in order to avoid over-provisioning cloud resources as much as possible.

Currently, the hosting bill for mybinder.org runs at a cost of around \$180 per day and around 7,000 users per day. This comes out to around $\frac{180 \times 30}{7000 \times 30} \approx 3$ cents per user. The mybinder.org team publishes its daily hosting costs in a public repository on GitHub [Jup18]. It hopes that this serves to encourage other organizations to deploy BinderHub for their own purposes, since it is possible to do so in a cost-effective manner.

Finally, because Kubernetes is an open source system for managing containers, it has been deployed on a number of cloud providers as well as on self-owned hardware and virtual machines.

5. <http://mybinder-sre.readthedocs.io/en/latest/>

6. <https://github.com/jupyterhub/binder-billing>

7. <https://grafana.mybinder.org>

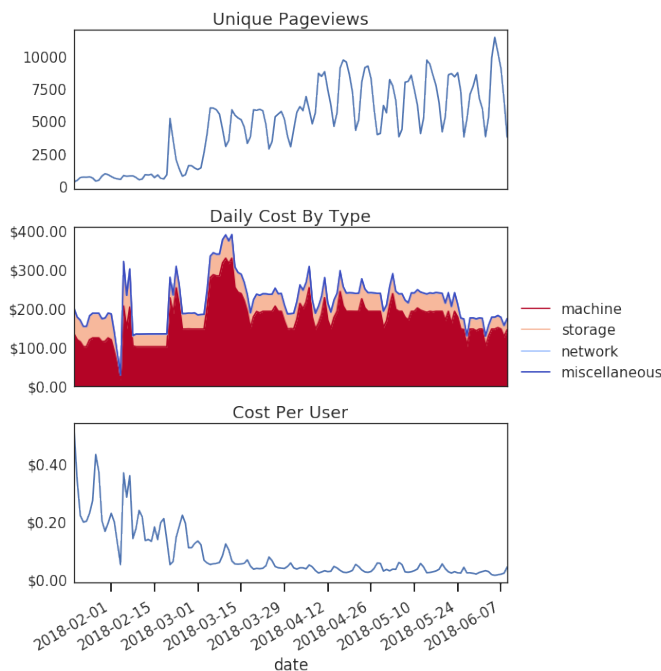


Fig. 5: Cloud computing costs for running *mybinder.org* in 2018. The x axis shows one point per day. The number of daily unique users has consistently grown over this time, while modifications to the BinderHub codebase (as well as the cloud resources used) have kept costs relatively flat. As a result, *mybinder.org* currently operates at about 3 cents per user per day.

While *mybinder.org* currently runs on the Google Cloud Platform, a BinderHub can run on any typical deployment of Kubernetes with minimal hardware requirements. This flexibility helps avoid vendor lock-in and is crucial for an open source tool such as BinderHub and JupyterHub. It also makes it possible for *mybinder.org* (or other BinderHub deployments) to seek the most cost-effective option for its needs.

Models for sustainability

The Binder team is exploring multiple models for sustaining the public digital infrastructure of *mybinder.org*, the team required to operate it, and the broader Binder ecosystem. At its current rate, the annual hosting cost of *mybinder.org* is around $180 \times 365 \approx \$66,000$, an amount that could be sustainable with a grant-funded model. Operating and supporting the public digital infrastructure of *mybinder.org* requires several staff members distributed globally to provide reasonable coverage across time zones for user support and incident response. This means salary costs will require a significant amount of funding.

The Binder team is actively exploring a *federation model* for BinderHub servers. Other organizations, companies, or universities can deploy their own BinderHubs for their own users or students, either on their own hardware or on cloud providers such as Google, Amazon, or Microsoft. These organization-specific deployments could require authentication or provide access to more complex cloud resources. In this case, *mybinder.org* could serve as a hub that connects this federated network of BinderHubs together, directing the user to an organization-specific BinderHub provided that they have the proper credentials on their machine.

The future of Binder

This paper outlines the technical infrastructure underlying *mybinder.org* and the BinderHub open source technology, including the guiding design principles and goals of the project. Binder is designed to be modular, to adapt itself to pre-existing tools and workflows in the open source community, and to be transparent in its development and operations.

Each of the tools described above is open source and permissively licensed, and we welcome the contributions and input from others in the open source community. In particular, we are excited to pursue Binder's development in the following scenarios:

- 1) **Reproducible publishing.** One of the core benefits of BinderHub is that it can generate deterministic environments that are linked to a code repository stored in a long-term archive like Zenodo⁸. This makes it useful for generating static representations of the environment needed to reproduce a scientific result. Binder has already been used alongside scientific publications ([LIG], [RHDV17], [CR18], [HRM⁺17], [RT16], [NHKvdW18]) to provide an interactive and reproducible document with minimal added effort. In the future, the Binder project hopes to partner with academic publishers and professional societies to incorporate these reproducible environments into the publishing workflow.
- 2) **Education and interactive materials.** Binder's goal is to lower the barrier to interactivity, and to allow users to utilize code that is hosted in repository providers such as GitHub. Because Binder runs as a free and public service, it could be used in conjunction with academic programs to provide interactivity when teaching programming and computational material. For example, the Foundations in Data Science course at UC Berkeley already utilizes *mybinder.org* to provide free interactive environments for its open source textbook. The Binder team hopes to find new educational uses for the technology moving forward.
- 3) **Access to complex cloud infrastructure.** While *mybinder.org* provides users with restricted hardware for cost-savings purposes, a BinderHub can be deployed on any cloud hardware that is desired. This opens the door for using BinderHub as a shared, interactive gateway that provides access to an otherwise inaccessible dataset or computational resource. For example, the GESIS Institute for Social Sciences provides a JupyterHub and BinderHub [GES] for their users at the university. The Binder team hopes to find new cases where BinderHub can be used as an entrypoint to provide individuals access to more sophisticated resources in the cloud.

Binder is a free, open source, and massively publicly available tool for easily creating sharable, interactive, reproducible environments in the cloud. The Binder team is excited to see the Binder community continue to evolve and utilize BinderHub for new uses in reproducibility and interactive computing.

REFERENCES

- [AESM17] André Anjos, Laurent El-Shafey, and Sébastien Marcel. BEAT: An Open-Source Web-Based Open-Science platform. April 2017. URL: <http://arxiv.org/abs/1704.02319>, arXiv:1704.02319.

8. <https://zenodo.org>

- [Atl] Atlassian. Bitbucket. <https://bitbucket.org>. Accessed: 2018-5-24. URL: <https://bitbucket.org>.
- [Bak16] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, May 2016. URL: <http://dx.doi.org/10.1038/533452a>.
- [BD95] Jonathan B Buckheit and David L Donoho. WaveLab and reproducible research. In Anestis Antoniadis and Georges Oppenheim, editors, *Wavelets and Statistics*, pages 55–81. Springer New York, New York, NY, 1995. URL: https://doi.org/10.1007/978-1-4612-2544-7_5.
- [Ber] Berkeley Division of Data Sciences. Foundations of data science. <http://data8.org/>. Accessed: 2018-5-23. URL: <http://data8.org/>.
- [Bus18] Matthias Bussonnier. I python, you r, we julia. <https://medium.com/@mbussonn/baf064ca1fb6>, April 2018. Accessed: 2018-5-23. URL: <https://medium.com/@mbussonn/baf064ca1fb6>.
- [CR18] Neil Cornish and Travis Robson. The construction and use of LISA sensitivity curves. March 2018. URL: <http://arxiv.org/abs/1803.01944>, [arXiv:1803.01944](https://arxiv.org/abs/1803.01944).
- [Doc] Docker, Inc. Docker. <https://www.docker.com/>. Accessed: 2018-5-24. URL: <https://www.docker.com/>.
- [FO16] Jeremy Freeman and Andrew Osheroff. Toward publishing reproducible computation with binder. <https://elifesciences.org/labs/a7d53a88/toward-publishing-reproducible-computation-with-binder>, May 2016. Accessed: 2017-12-11. URL: <https://elifesciences.org/labs/a7d53a88/toward-publishing-reproducible-computation-with-binder>.
- [GES] GESIS – Leibniz Institute for the Social Sciences. GESIS notebooks (beta). <https://notebooks.gesis.org/>. Accessed: 2018-5-23. URL: <https://notebooks.gesis.org/>.
- [Git] GitHub. GitHub. URL: <https://github.com>.
- [GL07] Robert Gentleman and Duncan Temple Lang. Statistical analyses and reproducible research. *J. Comput. Graph. Stat.*, 16(1):1–23, 2007. URL: <http://www.jstor.org/stable/27594227>.
- [Hea18] Tim Head. *openrefined*, 2018. URL: <https://github.com/betatim/openrefined>.
- [HRM⁺17] Christopher R Holdgraf, Jochem W Rieger, Cristiano Micheli, Stephanie Martin, Robert T Knight, and Frederic E Theunissen. Encoding and decoding models in cognitive electrophysiology. *Front. Syst. Neurosci.*, 11:61, September 2017. URL: <http://dx.doi.org/10.3389/fnsys.2017.00061>.
- [Jup18] JupyterHub. *binder-billing*, 2018. URL: <https://github.com/jupyterhub/binder-billing>.
- [LIG] LIGO Scientific Collaboration. LIGO open science center. <https://losc.ligo.org/tutorials/>. Accessed: 2017-12-12. URL: <https://losc.ligo.org/tutorials/>.
- [LV15] Percy Liang and Evelyne Viegas. CodaLab worksheets for reproducible, executable papers, December 2015. URL: <https://nips.cc/Conferences/2015/Schedule?showEvent=5779>.
- [Mic] Microsoft. Microsoft R application network. URL: <https://mran.microsoft.com/>.
- [Min] R K Min. *Thebelab*. <https://github.com/minrk/thebelab>. Accessed: 2018-6-13. URL: <https://github.com/minrk/thebelab>.
- [NHKvdW18] Mark C Neyrinck, Johan Hidding, Marina Konstantatou, and Rien van de Weygaert. The cosmic spiderweb: equivalence of cosmic, architectural and origami tessellations. *Royal Society Open Science*, 5(4):171582, April 2018. URL: <http://rsos.royalsocietypublishing.org/content/5/4/171582>.
- [nte16] nteract contributors. *nteract*, 2016. URL: <https://play.teract.io/>.
- [Ope] Open Humans Foundation. Personal data notebooks. <https://www.openhumans.org/activity/personal-data-notebooks/>. Accessed: 2018-5-24. URL: <https://www.openhumans.org/activity/personal-data-notebooks/>.
- [Pin17] Joelle Pineau. Reproducibility in deep reinforcement learning and beyond, December 2017. URL: <https://twitter.com/xtimv/status/938917013086380032>.
- [Pro17a] Project Jupyter Contributors. Using R with jupyter / RStudio on binder, 2017. URL: <https://github.com/binder-examples/r>.
- [Pro17b] Project Jupyter Contributors. *jupyterlab-demo*, 2017. URL: <https://github.com/jupyterlab/jupyterlab-demo>.
- [Pro17c] Project Jupyter Contributors. *repo2docker*, 2017. URL: <https://github.com/jupyter/repo2docker/>.
- [RHDV17] Andrew Slavlin Ross, Michael C Hughes, and Finale Doshi-Velez. Right for the right reasons: Training differentiable models by constraining their explanations. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages Pages 2662–2670., March 2017. URL: <https://www.ijcai.org/proceedings/2017/371>.
- [RN18] RK, Min and Daniel Nüst. *nbstencilaproxy*, 2018. URL: <https://github.com/minrk/nbstencilaproxy>.
- [RT16] Hanno Rein and Daniel Tamayo. Second-order variational equations for n-body simulations. *Monthly Notices of the Royal Astronomical Society*, 459(3):2275–2285, July 2016. URL: <https://academic.oup.com/mnras/article/459/3/2275/2595117>.
- [SHP12] V Stodden, C Hurlin, and C Pérignon. RunMyCode.org: A novel dissemination and collaboration platform for executing published computational results. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8, October 2012. URL: <http://dx.doi.org/10.1109/eScience.2012.6404455>.
- [Som18] James Somers. The scientific paper is obsolete. *The Atlantic*, April 2018. URL: <https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/>.
- [SSM18] Victoria Stodden, Jennifer Seiler, and Zhaokun Ma. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proc. Natl. Acad. Sci. U. S. A.*, 115(11):2584–2589, March 2018. URL: <http://dx.doi.org/10.1073/pnas.1708290115>.
- [The] The American Society for Cell Biology. ASCB member survey on reproducibility. Technical report. URL: <http://www.ascb.org/wp-content/uploads/2015/11/final-survey-results-without-Q11.pdf>.
- [Wik] Wikimedia. PAWS: A web shell. <https://wikitech.wikimedia.org/wiki/PAWS>. Accessed: 2018-5-23. URL: <https://wikitech.wikimedia.org/wiki/PAWS>.

Spatio-temporal analysis of socioeconomic neighborhoods: The Open Source Longitudinal Neighborhood Analysis Package (OSLNAP)

Sergio Rey^{‡*}, Elijah Knaap[‡], Su Han[‡], Levi Wolf[§], Wei Kang[‡]

https://youtu.be/VWMj_rNb0io



Abstract—The neighborhood effects literature represents a wide span of the social sciences broadly concerned with the influence of spatial context on social processes. From the study of segregation dynamics, the relationships between the built environment and health outcomes, to the impact of concentrated poverty on social efficacy, neighborhoods are a central construct in empirical work. From a dynamic lens, neighborhoods experience changes not only in their socioeconomic composition, but also in spatial extent; however, the literature has ignored the latter source of change. In this paper, we discuss the development of a novel, spatially explicit tool: the Open Source Longitudinal Neighborhood Analysis Package (OSLNAP) using the scientific Python ecosystem.

Index Terms—neighborhoods, GIS, clustering, dynamics

Introduction

For social scientists in a wide variety of disciplines, neighborhoods are central thematic topics, focal units of analysis, and first-class objects of inquiry. Despite their centrality in public health, sociology, geography, political science, economics, psychology, and urban planning, however, neighborhoods remain understudied. One of the reasons for that is because researchers lack appropriate analytical tools for understanding neighborhood evolution through time and space. Towards this goal we are developing the *open source longitudinal neighborhood analysis program* (OSLNAP). We envisage OSLNAP as a toolkit for better, more open and reproducible science focused on neighborhoods and their sociospatial ecology. In this paper we first provide an overview of the main components of OSLNAP. Next, we present an illustration of selected OSLNAP functionality. We conclude the paper with a road map for future developments.

OSLNAP

Neighborhood analysis involves a multitude of analytic tasks, and different types of inquiry lead to different analytical pipelines in which distinct tasks are combined in sequence. OSLNAP is designed in a modular fashion to facilitate the composition of

different pipelines for neighborhood analysis. Its functionality is available through several interfaces that include a web-based front end as well as a library for scripting in Jupyter notebooks or at the shell. As such, OSLNAP is intended to support different types of researchers and questions. For example, a sociologist interested in comparative segregation dynamics can use OSLNAP to derive time-consistent boundaries for a collection of US metropolitan areas from 1980-2010. Alternatively, public health epidemiologists can use the same boundaries to study the impact of neighborhood context on childhood obesity trends. Both of these types of studies might be characterized as "neighborhood effects" studies as neighborhood units serve as containers to study different socioeconomic processes.

An alternative group of studies falls under the "neighborhood dynamics" label. Here the interest is in the neighborhood units themselves and how their boundaries *and* internal socioeconomic composition evolve over time. Processes such as gentrification and the so called great inversion [Ehr12] where wealthy, higher educated, white populations are relocating into the center cities while growing numbers of minorities move to the suburbs both fundamentally restructure urban and suburban neighborhoods. OSLNAP is designed to support both neighborhood effects and neighborhood dynamics modes of inquiry.

Here we provide an overview of each of the main analytical components of OSLNAP before moving on to an illustration of how selections of the analytical functionality can be combined for particular use cases. OSLNAP's analytical components are organized into three core modules: [a] data layer; [b] neighborhood definition layer; [c] longitudinal analysis layer.

Data Layer

Like many quantitative analyses, one of the most important and challenging aspects of longitudinal neighborhood analysis is the development of a tidy and accurate dataset. When studying the socioeconomic makeup of neighborhoods over time, this challenge is compounded by the fact that the spatial units whose composition is under study often change size, shape, and configuration over time. The `harmonize` module provides social scientists with a set of simple and consistent tools for building transparent and reproducible spatiotemporal datasets. Further, the tools in `harmonize` allow researchers to investigate the implications of

* Corresponding author: sergio.rey@ucr.edu

‡ Center for Geospatial Sciences, University of California, Riverside

§ School of Geographical Sciences, University of Bristol



Fig. 1: Enumeration Unit Changes [U.S10].

alternative decisions in the data processing pipeline and how those decisions affect the results of their research.

Neighborhood demographic and socioeconomic data relevant to social scientists are typically collected via a household census or survey and aggregated to a geographic reporting unit such as a state, county or zip code which may be relatively stable. The boundaries of smaller geographies like census tracts, however, often are designed to encapsulate roughly the same number of people for the sake of comparability, which means that they are necessarily redrawn with each data release as population grows and fluctuates. Figure 1 illustrates the issues involved. Here two census tracts from 2000 have been merged to form a new tract in 2010. However, while one of the original tracts is completely contained in the new tracts, the second original tract is only partially contained in the new tract. In other words, since same physical location may fall within the boundary of different reporting units at different points in time, it is impossible to compare directly a single neighborhood with itself over time.

To facilitate temporal comparisons, research to date has proceeded by designating a “target” geographic unit or zone that is held constant over time, and allocating data from other zones using areal interpolation and other estimation techniques. This process is sometimes known as “boundary harmonization” [LSX16]. While “harmonized” data is used widely in neighborhood research, the harmonization process also has known shortcomings, since the areal interpolation of aggregate data is subject to the ecological fallacy—the geographic manifestation of which is known as the “Modifiable Areal Unit Problem” (MAUP) [Ope84]. Simply put, MAUP holds that areal interpolation introduces bias since the spatial distribution of variables in each of the overlapping zones is unknown. A number of alternative approaches have been suggested to reduce the amount of error by incorporating auxiliary data such as road networks, which help to uncover the “true” spatial distribution of underlying variables, but this remains an active area of research [Sch17], [SQ13], [Tap10], [Xie95].

In practice, these challenges mean that exceedingly few neighborhood researchers undertake harmonization routines in their own research, and those performing temporal analyses typically use exogenous, pre-harmonized boundaries from a commercial source such as the Neighborhood Change Database (NCDB) [Tat], or the freely available Longitudinal Tract Database (LTDB) [LXS14]. The developers of these products have published studies verifying the accuracy of their respective data, but those claims have gone untested because external researchers are unable to fully

replicate the underlying methodology.

To overcome the issues outlined above, OSLNAP provides a suite of methods for conducting areal interpolation and boundary harmonization in the `harmonize` module. It leverages `geopandas` and `PySAL` for managing data and performing geospatial operations, and the `PyData` stack for attribute calculations [RA10]. The `harmonize` module allows a researcher to specify a set of input data (drawn from the space-time database described in the prior section), a set of target geographic units to remain constant over time, and an interpolation function that may be applied to each variable in the dataset independently. For instance, a researcher may decide to use different interpolation methods for housing prices than for the share of unemployed residents, than for total population; not only because the researcher may wish to treat rates and counts separately, but also because different auxiliary information might be applicable for different types of variables.

In a prototypical workflow, `harmonize` permits the end-user to carry out a number of tasks: [a] compile and query a spatiotemporal database using either local data or connections to public data services; [b] define the relevant variables to be harmonized and optionally apply a different (spatial and/or temporal) interpolation function to each; [c] harmonize temporal data to consistent spatial units by either selecting an existing native unit (e.g. zip codes in 2016), inputting a user-defined unit (e.g. a theoretical or newly proposed boundary), or developing new primitive units (e.g. the intersection of all polygons).

Neighborhood Identification

Neighborhoods are complex social and spatial environments with multiple interacting individuals, markets, and processes. Despite decades of research it remains difficult to quantify neighborhood context, and certainly no single variable is capable of capturing the entirety of a neighborhood’s essential essence. For this reason, several traditions of urban research focus on the application of multivariate clustering algorithms to develop neighborhood typologies. Such typologies are sometimes viewed as more holistic descriptions of neighborhoods because they account for multiple characteristics simultaneously [Gal01].

One notable tradition from this perspective called “geodemographics”, is used to derive prototypical neighborhoods whose residents are similar along a variety of socioeconomic and demographic attributes [FG89], [SS14]. Geodemographics have been applied widely in marketing [FE05], education [SL09], and health research [PGL⁺11] among a wide variety of additional fields. The geodemographic approach has also been criticized, however, for failing to model geographic space formally. In other words, the geodemographic approach ignores spatial autocorrelation, or the “first law of geography”—that the attributes of neighboring zones are likely to be similar.

Another tradition in urban research, known as “regionalization” has thus been focused on the development of multivariate clustering algorithms that account for spatial dependence explicitly. To date, however, these traditions have rarely crossed in the literature, limiting the utility each approach might have toward applications in new fields. In the `cluster` module, we implement both clustering approaches to (a) foster greater collaboration among weakly connected components in the field of geographic information science, and (b) to allow neighborhood researchers to investigate the performance of multiple different clustering

solutions in their work and evaluate the implications of including space as a formal component in their clustering models.

In OSLNAP, the `cluster` module leverages the scientific python ecosystem, building from `scikit-learn` [PVG⁺11], `geopandas` [Geo18], and `PySAL` [Rey15]. Using input from the Data Layer, the `cluster` module allows researchers to develop neighborhood typologies based on either attribute similarity (the geodemographic approach) or attribute similarity with incorporated spatial dependence (the regionalization approach). Given a space-time data set, the `cluster` module permits three different treatments of time when defining neighborhoods. The first focuses on the case where only a single cross-section is available, and the clustering is carried out to define neighborhoods for that one point in time. In the second case, multiple waves or periods of observations are available and the clustering is repeated for each time slice of observations. This can be a useful approach if researchers are interested in the durability and permanence of certain kinds of neighborhoods. If similar types reappear in multiple cross sections (e.g. if the k-means algorithm places the k-centers in approximately similar locations each time period), then it may be inferred that the metropolitan dynamics are somewhat stable, at least at the macro level, since new kinds of neighborhoods do not appear to be evolving and old, established neighborhood types remain prominent. The drawback of this approach is the type of a single neighborhood cannot be compared between two different time periods because the types are independent in each period.

In the third approach, clusters are defined from all observations in all time periods. The universe of potential neighborhood types is held constant over time, the neighborhood types are consistent across time periods, and researchers can examine how particular neighborhoods get classified into different neighborhood types as their composition transitions through different time periods. While comparatively rare in the research, this latter approach allows a richer examination of socio-spatial dynamics. By providing tools to drastically simplify the data manipulation and analysis pipeline, we aim to facilitate greater exploration of urban dynamics that will help catalyze more of this research.

To facilitate this work, the `cluster` module provides wrappers for several common clustering algorithms from `scikit-learn` that can be applied. Beyond these, however, it also provides wrappers for several *spatial* clustering algorithms from `PySAL`, in addition to a number of state-of-the-art algorithms that have recently been developed [Wol18].

In a prototypical workflow, `cluster` permits the end-user to: [a] query the (tidy) space-time dataset created via the `harmonize` module; [b] define the neighborhood attributes and time periods and on which to develop a typology; [c] run one or more clustering algorithms on the space-time dataset to derive neighborhood cluster membership. Clustering may be applied cross-sectionally or on the pooled time-series, and clustering may incorporate spatial dependence, in which case `cluster` provides options for users to parameterize a spatial contiguity matrix. Clustering results may be reviewed quickly via the built-in `plot()` method, or interactively by leveraging the planned `geovisualization` module.

Longitudinal Analysis

Having identified the neighborhood types for all units of analysis over the whole time span, researchers might be interested in how they evolve over time. The third core module of OSLNAP's analytical components, `change`, provides a suite of functionality to-

wards this end. Traditional longitudinal analysis in neighborhood contexts focuses solely on changes in residential socioeconomic composition, while we and others have argued that changes in geographic footprints are also substantively interesting [RAF⁺11]. Therefore, this component draws upon recent methodological developments from spatial inequality dynamics and implements two broad sets of spatially explicit analytics to provide deeper insights into the evolution of socioeconomic processes and the interaction between these processes and geographic structure.

Both sets of analytics operate on time series of neighborhood types; they each take as input a set of spatial units of analysis (e.g. census tracts) that have been assigned a categorical variable for each point in time (e.g. the output of the `cluster` module). They differ, however, in how the time series are modeled and analyzed. The first set centers on *transition analysis*, which treats each time series as stochastically generated from time point to time point. It is in the same spirit of the first-order Markov Chain analysis where a (k,k) transition matrix is formed by counting transitions across all the k neighborhood types between any two consecutive time points for all spatial units. One drawback of this approach is that it treats all the time series as being independent of one another and following an identical transition mechanism. The spatial Markov approach was proposed by [Rey01] to interrogate potential spatial interactions by conditioning transition matrices on neighboring context while the spatial regime Markov approach allows several transition matrices to be formed for different spatial regimes which are constituted by contiguous spatial units. Both approaches together with inferences have been implemented in Python Spatial Analysis Library (`PySAL`) [Rey15] and Geospatial Distribution Dynamics (`giddy`) package [gid18]. The `change` module considers these packages as dependencies and wraps relevant classes and functions to make them consistent and efficient for longitudinal neighborhood analysis.

The other set of spatially explicit approach to neighborhood dynamics is concerned with *sequence analysis* which treats each time series of neighborhood types as a whole, in contrast to *transition analysis*. The core of *sequence analysis* is the similarity measure between a pair of sequences. Various aspects of a neighborhood sequence such as the order in which successive neighborhood types appears, the year(s) in which a specific neighborhood type appears, and the duration of a neighborhood type could be the focus of the similarity measure. Choosing which aspect or aspects to focus on should be driven by the research question at hand and the interpretation should proceed with caution [SR16]. A major approach of *sequence analysis*, the optimal matching (OM) algorithm, which was originally used for matching protein and DNA sequences [AT00], has been adopted to measure the similarity between neighborhood sequences in metropolitan areas such as Los Angeles and Chicago [Del16], [Del17]. It generally works by finding the minimum cost for transforming one sequence to another using a combination of operations including substitution, insertion, deletion and transposition. The similarity matrix is then used as the input for another round of clustering to derive a typology of neighborhood trajectory to produce several sequences of neighborhood types typically happening in a particular order [Del16].

In a prototypical workflow, the `change` module permits the end user to explore the nature of neighborhood change from a dynamic, holistic or combined holistic & dynamic perspective. From a dynamic perspective, *transition analysis* can be used to apply a first-order Markov chain model to look at probabilities

of transitioning between neighborhood types over time. It also supports the use of a spatial Markov chains model to interrogate the role of spatial interactions in shaping neighborhood dynamics or the application of a spatial regime Markov chains model to explore spatially heterogeneous neighborhood dynamics. From a holistic perspective, *sequence analysis* involves the application of the OM algorithm with classic cost functions for substitution, insertion, deletion and transposition, or those explicitly taking account of potential spatial dependence and spatial heterogeneity. Finally, a combined holistic & dynamic perspective is gained by feeding the output from *transiton analysis*, which is the empirical transition probability matrix, or spatially dependent transition probability matrices into *sequence analysis* to help set operation costs.

Empirical Illustration

In the following sections we demonstrate the utility of `OSLNAP` by presenting the results of several initial analyses conducted with the package. We begin with a series of cluster analyses, which are then used to analyze neighborhood dynamics. Typically, workflows of this variety would require extensive data collection, munging and recombination; with `OSLNAP`, however, we accomplish the same in just a few lines of code. Using the Los Angeles metropolitan area as our example, we present three neighborhood typologies, each of which leverages the same set of demographic and socioeconomic variables, albeit with different clustering algorithms. The results show similarities across the three methods but also several marked differences. This diversity of results can be viewed as either nuisance or flexibility, depending on the research question at hand, and highlights the need for research tools that facilitate rapid creation and exploration of different neighborhood clustering solutions. For each example, we prepare a cluster analysis for the Los Angeles metropolitan region using data at the census tract level. We visualize each clustering solution on a map, describe the resulting neighborhood types, and examine the changing spatial structure over time. For each of the examples, we cluster on the following variables: race categories (percent white, percent black, percent Asian, percent Hispanic), educational attainment (share of residents with a college degree or greater) and socioeconomic status (median income, median home value, percent of residents in poverty).

Agglomerative Ward

We begin with a simple example identifying six clusters via the agglomerative Ward method. Following the geodemographic approach, we aim to find groups of neighborhoods that are similar in terms of their residential composition, regardless of whether those neighborhoods are physically proximate. Initialized with the demographic and socioeconomic variables listed earlier, the Ward method identifies three clusters that are predominantly white on average but which differ with respect to socioeconomic status. The other three clusters, meanwhile, tend to be predominantly minority neighborhoods but are differentiated mainly by the dominant racial group (black versus Hispanic/Latino) rather than by class. The results, while unsurprising to most urban scholars, highlight the continued segregation by race and class that characterize American cities. For purposes of illustration, we give each neighborhood type a stylized moniker that attempts to summarize succinctly its composition (again, a common practice in the geodemographic literature). To be clear, these labels are oversimplifications of the

socioeconomic context within each type, but they help facilitate rapid consumption of the information nonetheless. The resulting clusters are presented in Figure 2.

- Type 0. racially concentrated (black and Hispanic) poverty
- Type 1. minority working class
- Type 2. integrated middle class
- Type 3. white upper class
- Type 4. racially concentrated (Hispanic) poverty
- Type 5. white working class

When the neighborhood types are mapped, geographic patterns are immediately apparent, despite the fact that space is not considered formally during the clustering process. These visualizations reveal what is known as “the first law of geography”—that near things tend to be more similar than distant things (stated otherwise, that geographic data tend to be spatially autocorrelated) [Tob70]. Even though we do not include the spatial configuration as part of the modeling process, the results show obvious patterns, where neighborhood types tend to cluster together in euclidian space. The clusters for neighborhoods type zero and four are particularly compact and persistent over time (both types characterized by racially concentrated poverty), helping to shed light on the persistence of racial and spatial inequality. With these types of visualizations in hand, researchers are equipped not only with analytical tools to understand how neighborhood composition can affect the lives of its residents (a research tradition known as neighborhood effects), but also how neighborhood identities can transform (or remain stagnant) over time and space. Beyond the simple diagnostics plots presented above, `OSLNAP` also includes an interactive visualization interface that allows users to interrogate the results of their analyses in a dynamic web-based environment where interactive charts and maps automatically readjust according to user selections.

Affinity Propagation

Affinity propagation is a newer clustering algorithm with implementations in `scikit-learn` that is capable of determining the number of clusters endogenously (subject to a few tuning parameters). Initialized with the default settings, `OSLNAP` discovers 14 neighborhood types in the Los Angeles region; in a way, this increases the resolution of the analysis beyond the Ward example, since increasing the number of clusters means neighborhoods are more tightly defined with lower variance in their constituent variables. On the other hand, increasing the number of neighborhood types also increase the difficulty of interpretation since the each type will be, by definition, less differentiable from the others. In the proceeding section, we discuss how researchers can exploit this variability in neighborhood identification to yield different types of dynamic analyses. Again, we find it useful to present stylized labels to describe each neighborhood type:

- Type 0. white working class
- Type 1. white extreme wealth
- Type 2. black working class
- Type 3. Hispanic poverty
- Type 4. integrated poverty
- Type 5. Asian middle class
- Type 6. white upper-middle class
- Type 7. integrated Hispanic middle class
- Type 8. extreme racially concentrated poverty
- Type 9. integrated extreme poverty

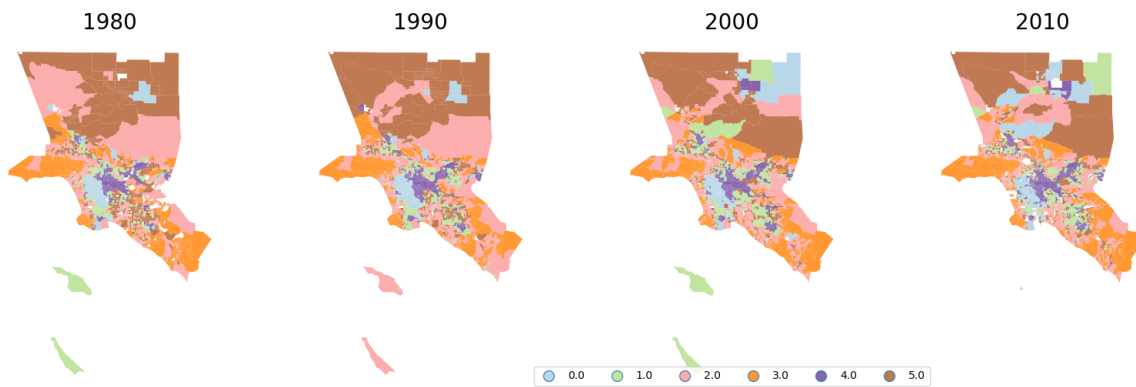


Fig. 2: Neighborhood Types in LA using Ward Clustering.

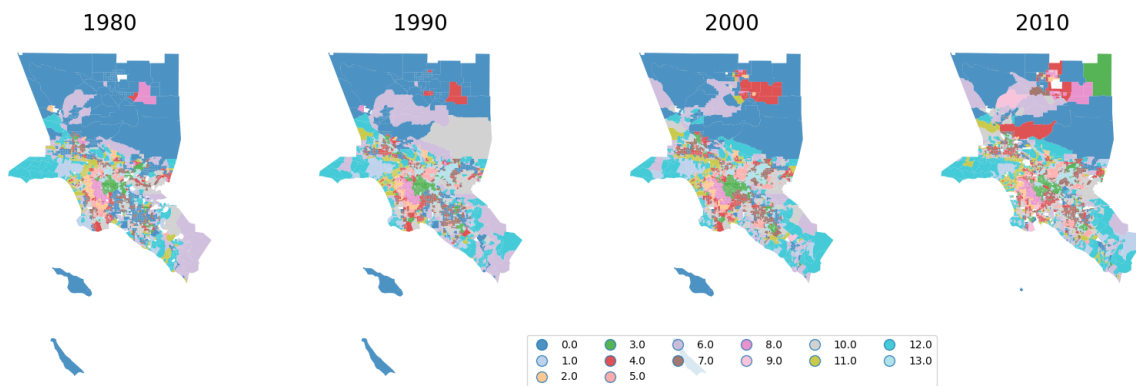


Fig. 3: Neighborhood Types in LA using Affinity Propagation.

- Type 10. Asian upper middle class
- Type 11. integrated white middle class
- Type 12. white elite
- Type 13. Hispanic middle class

Despite having more than double the number of neighborhood types in the Ward example, many of the spatial patterns remain when using affinity propagation clustering, including concentrated racial poverty in South Central LA, concentrated affluence along much of the coastline, black and Hispanic enclaves in the core of the city, and white working class strongholds in more rural areas to the north of the region. Comparing these two examples makes clear that some of the sociodemographic patterns in the LA region are quite stable, and are somewhat robust to the clustering method or number of clusters. Conversely, by increasing the number of clusters in the model, researchers can explore a much richer mosaic of social patterns and their evolution over time, such as the continued diversification of the I-5 corridor along the southern portion of the region.

SKATER

Breaking from the geodemographic approach, the third example leverages SKATER, a spatially-constrained clustering algorithm that finds groups of neighborhoods that are similar in composition, but groups them together if and only if they also satisfy the criteria

for a particular geographic relationship [Wol18]. As such, the family of clustering algorithms that incorporate spatial constraints (from the tradition known as “regionalization”) must be applied cross-sectionally, and yield an independent set of clusters for each time period, as shown in Figure 4. The clusters, thus, depend not only on the composition of the census units, but also their spatial configuration and connectivity structure at any given time.

Despite the fact that clusters are independent from one year to the next (and thus, we lack appropriate space in this text for describing the SKATER results for each year) comparing the results over time nonetheless yield some interesting insights. Regardless of the changing spatial and demographic structure of the Los Angeles region, some of the neighborhood boundaries identified are remarkably stable, such as the area of concentrated affluence in Beverly Hills and its nearby communities that jut out to the region’s West. Conversely, there is considerable change among the predominantly minority communities in the center of the region, whose boundaries appear to be evolving considerably over time. In these places, a researcher might use the output from SKATER to conduct an analysis to determine the ways in which the empirical neighborhood boundaries derived from SKATER conform to residents’ perceptions of such boundaries, their evolution over time, and their social re-definition as developed by different residential groups [Wol18]. Irrespective of its

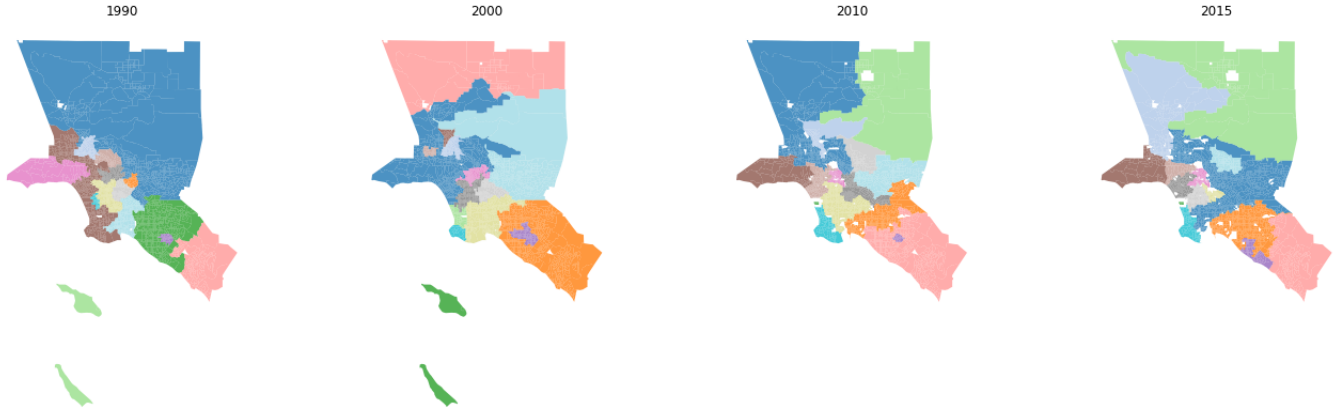


Fig. 4: Neighborhood Types in LA using SKATER.

particular use, the regionalization approach presents neighborhood researchers with another critical tool for understanding the bi-directional relationship between people and places.

In each of the sample analyses presented above, we use OSLNAP to derive a set of neighborhood clusters or types that can be used to analyze the demographic makeup of places over time. In some cases, these maps can serve as foundations for descriptive analyses or be analyzed as research projects in their own right. In other cases, in which social processes rather than the demographic makeup of communities are the focus of study, the neighborhood types derived here can be used as input to dynamic analyses of neighborhood change and evolution, particularly as they relate to phenomena such as gentrification and displacement. In the following sections, we demonstrate how the neighborhood typologies generated by OSLNAP's `cluster` module can be used as input to the `change` module to explore the neighborhood evolution.

Transition Analysis to Neighborhood Change

The `change` module can provide insights into the nature of neighborhood change in the Los Angeles metropolitan area. We utilize the neighborhood types for all census tracts of the Los Angeles metropolitan area across four census years identified by selected clustering algorithms in the former section as the input for the `change` module. Among the three clustering algorithms, SKATER was applied to each cross section of census tracts independently yielding clusters which are not directly comparable over time. Thus, we focus only on the six neighborhood types identified by the agglomerative Ward method (Fig. 2) and the fourteen neighborhood types identified by the affinity propagation method (Fig. 3).

We start with the aspatial transition analysis which pools all the time series of neighborhood types and counts how many transitions between any pair of neighborhood types across immediate consecutive census years $(t, t + 10)$ (or $(t, t + 5)$ for 2010-2015) which are further organized into a (k, k) transition count matrix \mathbf{N} . Adopting the maximum likelihood estimator for the first-order Markov transition probability as shown in Equation (1), a (k, k) transition probability matrix can thus be constructed providing the insights in the underlying dynamics of neighborhood change. The $(6, 6)$ and the $(14, 14)$ transition probability matrices for Ward and affinity propagation clusters are estimated and visualized in

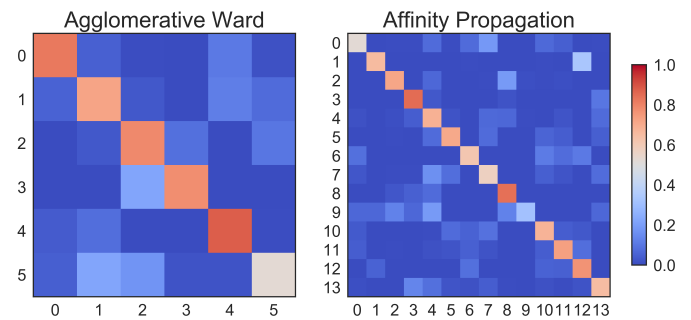


Fig. 5: Markov transition probability matrix for Ward and Affinity Propagation clusters.

Fig. 5 where the color in grid (i, j) represents the probability of transitioning from neighborhood type i to j in the next census year. It is obvious that both transition probability matrices are characterized by large diagonal entries, indicating a certain level of neighborhood stability for the focal four census years. This is especially true for the Ward neighborhood type 4 which is characterized by racially concentrated (Hispanic) poverty. The probability of staying at this type is 0.876 meaning that there is only 12.4% chance of changing to other neighborhood types once the census tract enters into type 4.

$$\hat{p}_{ij} = \frac{n_{ij}}{\sum_{q=1}^k n_{iq}}, \quad \text{where } i, j \in \mathbb{S} = \{1, 2, \dots, k\} \quad (1)$$

Moving from the aspatial transition analysis, we interrogate potential spatial interactions among neighborhood dynamics using the spatial Markov chain approach. More specifically, we hypothesize that the transition probability for any focal census tract is not constant, but rather dependent on the spatial context, that is, the most common neighborhood type of contiguous tracts, the so-called spatial lag. Therefore, k exhaustive and mutually exclusive subsamples are constructed based on the spatial lag at t , from which k (k, k) transition probability matrices are estimated based on Equation (1). Fig. 6 displays the spatial Markov transition probability matrices for Ward neighborhood types. It should be noted that the interpretation with these conditional transition probabilities should proceed with caution as the increased number of parameters to be estimated here could lead to large standard errors for some estimates. For example, the $(0, 0)$ entry in the

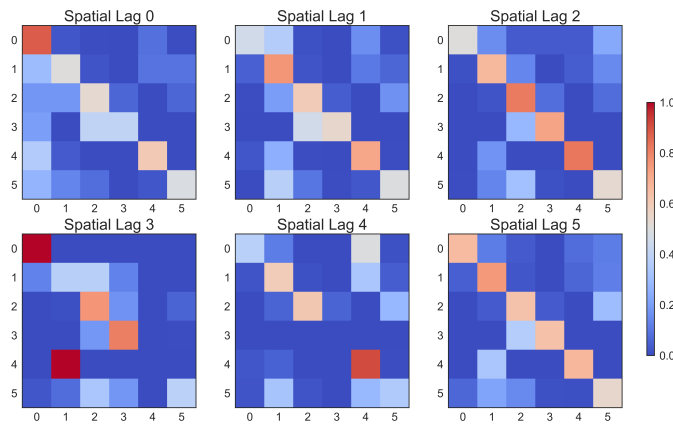


Fig. 6: Spatial Markov transition probability matrices for Ward clusters.

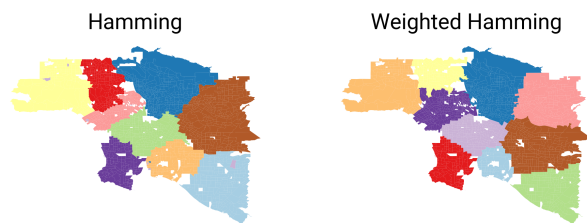


Fig. 7: Neighborhoods with similar spatial-social histories since 1980

subplot of Spatial Lag 3 is 1. The tendency of interpreting the 100 percent to be tracts "perfectly stuck at" Ward neighborhood type 0 if the spatial lag is type 3 should be compromised by the fact that there is only 1 observation transitioning from type 0 which has the spatial lag of type 3 at t and this very observation happens to stay at type 0. Since we are short of information, we could not conclude with the "perfectly stuck" theory. The spatial Markov tests (available upon request) including the likelihood ratio test and the χ^2 test [BB03], [RKW16] are both rejected indicating that neighboring context plays an important role in shaping the neighborhood dynamics.

Sequence Analysis to Neighborhood Change

Armed with the sequences of sociodemographic classifications for every harmonized tract in LA, the distance between these sequences can be computed. Since these sequences are intrinsically aligned in time, the Hamming distance between classifications yields an effective metric for how different places' demographic changes have been. The pairwise Hamming distance matrix for demographic transitions in LA is sufficient to recover a set of boundaries. However, alone, this metric only considers that two areas are in different sociodemographic classifications at a specific point in time. It does not consider the difference in the attribute's strength of assignment in these classifications, nor does it consider how well an area fits into its demographic classification.

Conceptually, this is important; even though the gist of the demographic classifications stay consistent over time, the members of these classes may shift around significantly over time. As a tract drifts from one classification to another classification over time, it may move within the class before it hops classifications

if the movement is slow. This means that, at each point in time, tracts are more or less representative of their clusters; a transition of one area from "white working class" to "white upper class" may not necessarily reflect the same amount of social/spatial volatility as a move from "minority working class" to "white upper class," as might happen during rapid gentrification.

As such, we can also weight the edit distance based on how "expensive" the edit is in terms of the clustering distance. Using this weighting method, not all transitions from white working class to white upper class will be treated the same: observations that are "almost" white upper class but not quite will be considered more similar to white upper class tracts. But, since a reassignment is still involved, there will still be a cost associated with that edit. Clusterings for both the raw Hamming edit distance and the weighted Hamming edit distances over sociodemographic sequences are shown in Figure 7 using [Wol18]. Broadly speaking, the assignments between the two clustering methods are strongly related (with an adjusted Rand index of .68), but macro-level distinctions between assignment structures are visible, particularly in the areas of central northern LA near the Hollywood Hills, as well as the areas of east LA, near Fullerton. This means that, when the sub-classification information is taken into account, clusterings can change. However, when examining spatially-contiguous clusters, the total amount of possible change is often quite constrained as well. Thus, the move from unweighted to weighted edit distances may make even more of a difference in some cases.

Future Directions

At present, we are in the early phases of the project and moving forward we will be focusing on the following directions.

Parameter sweeps: In the definition of neighborhoods, a researcher faces a daunting number of decisions surrounding treatment of harmonization, selection of variables, and choice of clustering algorithm, among others. In the neighborhood literature, the implications of these decisions remain unexplored and this is due to the computational burdens that have precluded formal examination. We plan on a modular design for OSLNAP that would support extensive parameter sweeps to provide an empirical basis for exploring these issues and to offer applied researchers computationally informed guidance on these decisions.

Data services: OSLNAP is being designed to work with existing harmonized data sets available from various firms and research labs. Because these fall under restrictive licenses, users must first acquire these sources - they cannot be distributed with OSLNAP. To address the limitations associated with this strategy, we are exploring interfaces to public data services such as CenPy [cen18] and tigris [tig18].

Interactive visualization: Apart from scripted environments demonstrated in this paper, OSLNAP is being designed with a web-based, interactive front-end that allows users to explore the results of different neighborhood analyses with the assistance of linked maps, charts, and tables. Together, these linked "views" allow a researcher to interrogate their results in a manner far richer than creating a series of static maps.

Reproducible Urban Data Science: A final direction for future research is the development of reproducible workflows as part of OSLNAP. Here we envisage leveraging our earlier work on provenance for spatial analytical workflows [ARL14] and extending it to the full longitudinal neighborhood analysis pipeline.

Conclusion

In this paper we have presented the motivation for, initial design, and implementation of OSLNAP. We feel that, even at this early stage in the project, OSLNAP has benefitted from the scope and deep nature of the PyData stack as we have been able to move from conceptualization to prototyping in fairly short order. At the same time, we see OSLNAP playing an important role in widening the use of Python in urban and spatial data science. We are looking forward to the future development of OSLNAP and interaction with both the PyDATA community and the broader community of computational social sciences.

Acknowledgment

This research was supported by NSF grant SES-1733705.

REFERENCES

- [ARL14] Luc Anselin, Sergio J. Rey, and Wenwen Li. Metadata and provenance for spatial analysis: the case of spatial weights. *International Journal of Geographical Information Science*, 28(11):2261–2280, May 2014. doi:10.1080/13658816.2014.917313.
- [AT00] Andrew Abbott and Angela Tsay. Sequence analysis and optimal matching methods in sociology: Review and prospect. *Sociological Methods & Research*, 29(1):3–33, 2000. doi:10.1177/0049124100029001001.
- [BB03] F. Bickenbach and E. Bode. Evaluating the Markov property in studies of economic convergence. *International Regional Science Review*, 26(3):363–392, 2003. doi:10.1177/0160017603253789.
- [cen18] cenpy Developers. cenpy. <https://github.com/ljwolf/cenpy>, 2018.
- [Del16] Elizabeth C Delmelle. Mapping the DNA of urban neighborhoods: Clustering longitudinal sequences of neighborhood socioeconomic change. *Annals of the American Association of Geographers*, 106(1):36–56, 2016. doi:10.1080/00045608.2015.1096188.
- [Del17] Elizabeth C Delmelle. Differentiating pathways of neighborhood change in 50 U.S. metropolitan areas. *Environment and Planning A*, 49(10):2402–2424, oct 2017. doi:10.1177/0308518X17722564.
- [Ehr12] Alan Ehrenhalt. *The great inversion and the future of the American city*. Random House, 2012.
- [FE05] Marc Farr and Andy Evans. Identifying ‘unknown diabetics’ using geodemographics and social marketing. *Journal of Direct, Data and Digital Marketing Practice*, 7(1):47–58, aug 2005. doi:10.1057/palgrave.ddmp.4340504.
- [FG89] R Flowerdew and W Goldstein. Geodemographics in Practice: Developments in North America. *Environment and Planning A*, 21(5):605–616, may 1989. doi:10.1068/a210605.
- [Gal01] George Galster. On the Nature of Neighbourhood. *Urban Studies*, 38(12):2111–2124, nov 2001. doi:10.1080/00420980120087072.
- [Geo18] GeoPandas Developers. GeoPandas 0.3.0. <http://geopandas.org/index.html>, 2018.
- [gid18] giddy Developers. Geospatial Distribution DYNamicis. <http://github.com/pysal/giddy.html>, 2018.
- [LSX16] John R. Logan, Brian J. Stults, and Zengwang Xu. Validating population estimates for harmonized census tract data, 2000–2010. *Annals of the American Association of Geographers*, 106(5):1013–1029, Jun 2016. doi:10.1080/24694452.2016.1187060.
- [LXS14] John R. Logan, Zengwang Xu, and Brian J. Stults. Interpolating U.S. decennial census tract data from as early as 1970 to 2010: A longitudinal tract database. *The Professional Geographer*, 66(3):412–420, May 2014. doi:10.1080/00330124.2014.905156.
- [Ope84] S Openshaw. Ecological Fallacies and the Analysis of Areal Census Data. *Environment and Planning A*, 16(1):17–31, jan 1984. doi:10.1068/a160017.
- [PGL⁺11] Jakob Petersen, Maurizio Gibin, Paul Longley, Pablo Mateos, Philip Atkinson, and David Ashby. Geodemographics as a tool for targeting neighbourhoods in public health campaigns. *Journal of Geographical Systems*, 13(2):173–192, 2011. doi:10.1007/s10109-010-0113-9.
- [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.
- [RA10] Sergio J. Rey and Luc Anselin. PySAL: A Python Library of Spatial Analytical Methods. In *Handbook of Applied Spatial Analysis*, volume 37, pages 175–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-03647-7_11.
- [RAF⁺11] Sergio J. Rey, Luc Anselin, David C. Folch, Daniel Arribas-Bel, Myrna L. Sastré Gutiérrez, and Lindsey Interlante. Measuring Spatial Dynamics in Metropolitan Areas. *Economic Development Quarterly*, 25(1):54–64, feb 2011. doi:10.1177/0891242410383414.
- [Rey01] S. J. Rey. Spatial empirics for economic growth and convergence. *Geographical Analysis*, 33(3):195–214, 2001. doi:10.1111/j.1538-4632.2001.tb00444.x.
- [Rey15] Sergio J. Rey. Python Spatial Analysis Library (PySAL): An update and illustration. In Chris Brunsdon and Alex Singleton, editors, *GeoComputation: A Practical Primer*, pages 233–253. SAGE Publications Ltd, 2015. doi:10.1007/978-3-642-03647-7_11.
- [RKW16] Sergio J. Rey, Wei Kang, and Levi Wolf. The properties of tests for spatial effects in discrete markov chain models of regional income distribution dynamics. *Journal of Geographical Systems*, 18(4):377–398, 2016. doi:10.1007/s10109-016-0234-x.
- [Sch17] Jonathan P. Schroeder. Hybrid areal interpolation of census counts from 2000 blocks to 2010 geographies. *Computers, Environment and Urban Systems*, 62:53–63, Mar 2017. doi:10.1016/j.compenvurbsys.2016.10.001.
- [SL09] Alexander D Singleton and Paul A Longley. Creating open source geodemographics: Refining a national classification of census output areas for applications in higher education. *Papers in Regional Science*, 88(3):643–666, aug 2009. doi:10.1111/j.1435-5957.2008.00197.x.
- [SQ13] Harini Sridharan and Fang Qiu. A Spatially Disaggregated Areal Interpolation Model Using Light Detection and Ranging-Derived Building Volumes. *Geographical Analysis*, 45(3):238–258, jul 2013. doi:10.1111/gean.12010.
- [SR16] Matthias Studer and Gilbert Ritschard. What matters in differences between life trajectories: a comparative review of sequence dissimilarity measures. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 179(2):481–511, 2016. doi:10.1111/rssa.12125.
- [SS14] Alexander D Singleton and Seth E Spielman. The Past, Present, and Future of Geodemographic Research in the United States and United Kingdom. *The Professional Geographer*, 66(4):558–567, oct 2014. doi:10.1080/00330124.2013.848764.
- [Tap10] Anna F. Tapp. Areal Interpolation and Dasymetric Mapping Methods Using Local Ancillary Data Sources. *Cartography and Geographic Information Science*, 37(3):215–228, 2010. doi:10.1559/152304010792194976.
- [Tat] Peter Tatian. Local scene: Neighborhood change database (ncdb). *PsycEXTRA Dataset*. doi:10.1037/e479172006-003.
- [tig18] tigris Developers. tigris. <https://github.com/walkerke/tigris>, 2018.
- [Tob70] W. R. Tobler. A computer movie simulating urban growth in the Detroit region. *Economic Geography*, 46(2):234–240, 1970. doi:10.2307/143141.
- [U.S10] U.S. Census. *Understanding the 2010 Tract Relationship Files*, 2010. URL: <https://www2.census.gov/geo/pdfs/maps-data/data/rel/tractrelfile.pdf>.
- [Wol18] Levi John Wolf. Spatially-Encouraged Spectral Clustering : A Critical Revision of Spatially-Constrained Spectral Clustering. 2018.
- [Xie95] Yichun Xie. The overlaid network algorithms for areal interpolation problem. *Computers, Environment and Urban Systems*, 19(4):287–306, 1995. doi:10.1016/0198-9715(95)00028-3.

Design and Implementation of pyPRISM: A Polymer Liquid-State Theory Framework

Tyler B. Martin^{¶*}, Thomas E. Gartner III[‡], Ronald L. Jones[¶], Chad R. Snyder[¶], Arthi Jayaraman^{‡§}

<https://youtu.be/MYw-pmz02p0>

Abstract—In this work, we describe the code structure, implementation, and usage of a Python-based, open-source framework, pyPRISM, for conducting polymer liquid-state theory calculations. Polymer Reference Interaction Site Model (PRISM) theory describes the equilibrium spatial-correlations, thermodynamics, and structure of liquid-like polymer systems and macromolecular materials. pyPRISM provides data structures, functions, and classes that streamline predictive PRISM calculations and can be extended for other tasks such as the coarse-graining of atomistic simulation force-fields or the modeling of experimental scattering data. The goal of providing this framework is to reduce the barrier to correctly and appropriately using PRISM theory and to provide a platform for rapid calculations of the structure and thermodynamics of polymeric fluids and polymer nanocomposites.

Index Terms—polymer, materials science, modeling, theory

Introduction

Free and open-source (FOSS) scientific software lowers the barriers to applying theoretical techniques by codifying complex approaches into usable tools that can be leveraged by non-experts. Here, we describe the implementation and structure of pyPRISM, a Python tool which implements Polymer Reference Interaction Site Model (PRISM) theory. [MGIJ⁺18], [dis] PRISM theory is an integral equation formalism that describes the structure and thermodynamics of polymer liquids. [SC87] Despite the successful application of PRISM theory to study a variety of complex soft-matter systems, [SC94] its use has been limited compared to other theory and simulation methods that have available open-source tools, such as Self-Consistent Field Theory (SCFT), [psc], [AQM⁺16] Molecular Dynamics (MD), [hoo], [GNA⁺15], [ALT08], [lam], [Pli95] or Monte Carlo (MC), [sim], [cas], [RM11]. Some important factors contributing to this reduced usage are the complexities associated with implementing PRISM theory and the lack of an available open-source codebase. Our previous publication, [MGIJ⁺18], focused primarily on the theoretical aspects of the method and presented several case studies to illustrate the utility of PRISM theory. In this work, we focus more specifically on the practical implementation and usage of PRISM theory within the pyPRISM framework. In the following

* Corresponding author: tyler.martin@nist.gov

¶ National Institute of Standards and Technology

‡ Chemical and Biomolecular Engineering, University of Delaware

§ Materials Science and Engineering, University of Delaware

Copyright © 2018 Tyler B. Martin et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

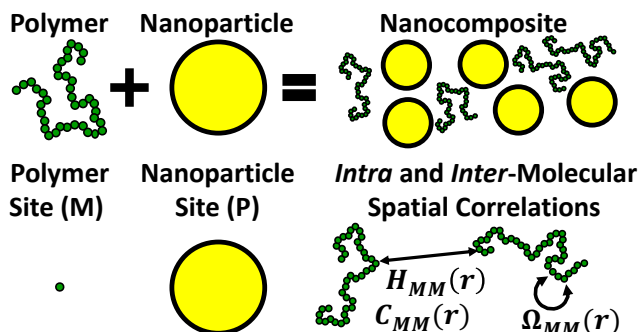


Fig. 1: A schematic representation of the components of a coarse grained polymer nanocomposite made up of polymer chains and large spherical nanoparticles. This system is the focus of reference [HS05]. In this example, there are two site-types: a monomer site-type (M) in green and a nanoparticle site-type (P) in yellow. Also labeled are the polymer-polymer intra-molecular ($\Omega_{M,M}(r)$) and inter-molecular correlation functions ($H_{M,M}(r)$ and $C_{M,M}(r)$).

sections, we will briefly discuss the basics of PRISM theory, our implementation of the theory in pyPRISM, our approach toward educating the scientific community about PRISM theory and pyPRISM, and finally our view for the future of the tool.

PRISM Theory

For a detailed discussion of PRISM theory, as well as a review of key applications of the theory, we direct the reader to our previous publication. [MGIJ⁺18] Here, we briefly highlight the salient points of PRISM theory in order to help motivate the design of our class structure.

PRISM theory describes the *spatial correlations* in a liquid-like polymer system made up of spherical interacting "sites." The category of liquid-like polymers includes melts, blends, solutions, and nanocomposites of both homopolymers and copolymers. Within these systems, PRISM is able to handle varying chain chemistry, monomer sequence, and topology. The traditional PRISM formalism is spherically symmetric, which in general prevents the use of PRISM to study glassy, crystalline, phase-separated or otherwise non-isotropic materials. While there is a modified PRISM formalism for oriented (liquid-crystalline) materials, [OS05a], [OS05b], [PS00], [PS99] those modifications are outside the scope of the current work. Figure 1 shows a schematic

of a polymer nanocomposite that could be studied with PRISM theory using a two-site model.

In general, PRISM sites represent a segment of a molecule or polymer chain, similar to the atoms or coarse-grained beads that comprise an MD or MC simulation. Unlike these simulation methods, PRISM treats all of the sites of a given type as indistinguishable and does not track the individual positions of each site in space. Instead, the structure of the system is described through average spatial correlation functions. The fundamental PRISM equation for multi-component systems is represented in Fourier-space as a matrix equation of the site-site spatial correlation functions.

$$\hat{H}(k) = \hat{\Omega}(k)\hat{C}(k) [\hat{\Omega}(k) + \hat{H}(k)] \quad (1)$$

In this expression, $\hat{H}(k)$ is the *inter*-molecular total correlation function matrix, $\hat{C}(k)$ is the *inter*-molecular direct correlation function matrix, and $\hat{\Omega}(k)$ is the *intra*-molecular correlation function matrix. $\hat{\Omega}(k)$ describes the how the monomers *within a molecule* are connected and placed, $\hat{H}(k)$ and $\hat{C}(k)$ describe how the molecules are placed in space relative to one another. The key difference between $\hat{H}(k)$ and $\hat{C}(k)$ is that the former includes many-body effects, while the latter does not. Knowledge of $\hat{H}(k)$, $\hat{C}(k)$, and $\hat{\Omega}(k)$ for a given system allows one to calculate a range of important structural and thermodynamic parameters, e.g., structure factors, radial distribution functions, second virial coefficients, Flory-Huggins χ parameters, bulk isothermal compressibilities, and spinodal decomposition temperatures.

Each of the variables in Equation 1 represents a function of wavenumber k which returns an $n \times n$ matrix, with n being the number of site-types in the calculation. Each element of a correlation function matrix (e.g., $\hat{H}_{\alpha,\beta}(k)$) represents the value of that correlation function between site types α and β at a given wavenumber k . These correlation function matrices are symmetric, therefore there are $\frac{n(n+1)}{2}$ independent site-type pairs and correlation function values in each correlation function matrix. The nanocomposite in Figure 1 is modeled using $n = 2$ site-types which yields three independent site-type pairs: polymer-polymer, polymer-particle, and particle-particle.

Equation 1, as written, has one unspecified degree of freedom for each site-type pair, therefore additional mathematical relationships must be supplied to obtain a solution. These relationships are called closures and are derived in various ways from fundamental liquid-state theory. Closures are also how the chemistry of a system is specified *via* pairwise interaction potentials $U_{\alpha,\beta}(r)$. For example, one widely-used closure is the Percus-Yeivick closure shown below

$$C_{\alpha,\beta}(r) = \left(e^{-U_{\alpha,\beta}(r)} - 1.0 \right) \left(1.0 + \Gamma_{\alpha,\beta}(r) \right), \quad (2)$$

where $\Gamma(r)$ is defined in real-space as

$$\Gamma_{\alpha,\beta}(r) = H_{\alpha,\beta}(r) - C_{\alpha,\beta}(r). \quad (3)$$

While the PRISM equation can be solved analytically [SC94] in select cases, we focus on a more generalizable numerical approach here. Figure 2 shows a schematic of our approach. For all site-types or site-type pairs, the user provides input values for $\hat{\Omega}_{\alpha,\beta}(k)$, site-site pair potentials $U_{\alpha,\beta}(r)$, site-type densities ρ_α , and an initial guess for all $\Gamma_{\alpha,\beta}(r)$. After the user supplies all necessary parameters and input correlation functions, pyPRISM applies a numerical optimization routine, such as a Newton-Krylov method, [new] to minimize a self-consistent cost function. The details of this cost function were discussed in our previous work. [MGJ⁺18]

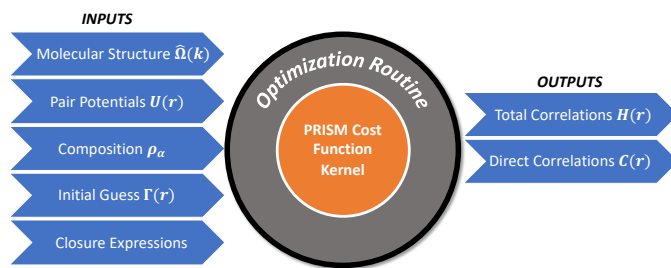


Fig. 2: Schematic of PRISM theory numerical solution process.

After the cost function is minimized, the PRISM equation is considered "solved" and the resultant correlation functions can be used for calculations.

pyPRISM Overview

pyPRISM defines a scripting API (application programming interface) that allows users to conduct calculations and numerically solve the PRISM equation for a range of liquid-like polymer systems. All of the theoretical details of PRISM theory are encapsulated in classes and methods which allow users to specify parameters and input correlation functions by name e.g., `PercusYeivick` for Equation 2. Furthermore, the structure of these classes was kept as simple as possible so that novice scientific programmers could easily extend pyPRISM by implementing new closures, potentials, or intra-molecular correlation functions. This code structure of pyPRISM is shown schematically in Figure 3 and is discussed in the [Implementation](#) Section.

Providing a scripting API rather than an "input file"-based scheme gives users the ability to use the full power of Python for complex PRISM-based calculations. For example, one could use parallelized loops to fill a database with PRISM results using Python's built-in support for thread or process pools. Alternatively, pyPRISM could easily be coupled to a simulation engine by calling the engine *via* a subprocess, processing the engine output, and then feeding that output to a pyPRISM calculation. The pyPRISM API is demonstrated in the [Example pyPRISM Script](#) section by modeling the system shown in Figure 1.

While experts in PRISM theory likely will need little guidance on how to appropriately apply pyPRISM, we also would like to make pyPRISM accessible to the widest possible audience. To this end, we have created comprehensive documentation [pyPa] and tutorial [pyPb] materials. Users can also try pyPRISM in their web-browser by visiting [pyPc]. See the [Pedagogy](#) section for more information on our philosophy in educating the scientific community about pyPRISM.

Installation

pyPRISM is a Python library that has been tested on Linux, OS X, and Windows with the CPython 2.7, 3.5 and 3.6 interpreters and only depends on Numpy [num], [WCV11] and Scipy [sci], [Oli07] for core functionality. Optionally, pyPRISM provides a unit conversion utility if the Pint [pin] library is available and a simulation trajectory analysis tool if pyPRISM is compiled with Cython [cyt]. pyPRISM is available on GitHub, [pyPd], conda-forge [pyPe] and the Python Package Index (PyPI) [pyPf] for download. It can be installed from the command line *via*

```
$ conda install -c conda-forge pyPRISM
```

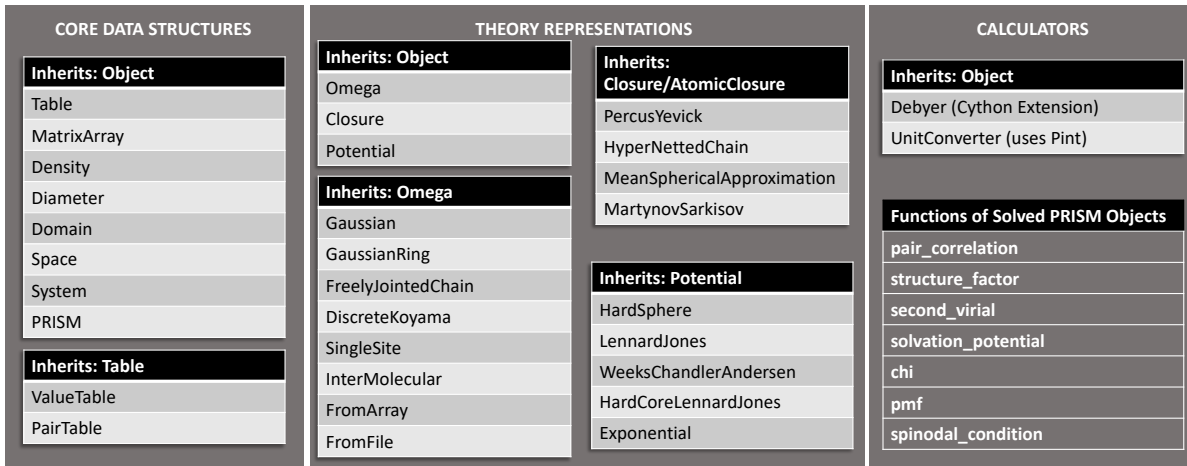


Fig. 3: Overview of codebase and class organization. A full description of the codebase classes and methods can be found in the online documentation. [pyPa].

or alternatively

```
$ pip install pyPRISM
```

Full installation instructions can be found in the documentation. [pyPa]

Implementation

Figure 3 shows an overview of the available classes and functions in pyPRISM and how they relate categorically. To begin, we consider the core data structures listed in the left column of the figure. Parameters and data in PRISM theory fall into two categories: those that define the properties of a single site-type (e.g., density, diameter) and those that define properties for a site-type pair (e.g., closure, potential, intra-molecular correlation functions). pyPRISM defines two base container classes based on this concept, both of which inherit from a parent pyPRISM.Table class: pyPRISM.ValueTable and pyPRISM.PairTable. These classes store numerical and non-numerical data, support complex iteration, and provide a .check() method that is used to ensure that all parameters are fully specified. Both pyPRISM.Table subclasses also support setting multiple pair-data at once, thereby making scripts easier to maintain via reduced visual noise and repetition. Additionally, pyPRISM.ValueTable automatically invokes matrix symmetry when a user sets an off-diagonal pair, assigning the α, β and β, α pairs automatically.

```
1 '''
2 Example of pyPRISM.ValueTable usage
3 '''
4
5 import pyPRISM
6
7 PT = pyPRISM.PairTable(types=['A', 'B', 'C'],
8                       name='potential')
9
10 # Set the A-A pair
11 PT['A', 'A'] = 'Lennard-Jones'
12
13 # Set the B-A, A-B, B-B, B-C, and C-B pairs
14 PT['B', ['A', 'B', 'C']] = 'Weeks-Chandler-Andersen'
15
16 try:
17     # Raises ValueError b/c not all pairs are set
18     PT.check()
19 except ValueError:
20     print('Not all pairs are set in ValueTable!')
```

```
22 # Set the C-A, A-C, C-C pairs
23 PT['C', ['A', 'C']] = 'Exponential'
24
25 # No-op as all pairs are set
26 PT.check()
27
28 for i,t,v in PT.iterpairs():
29     print('{} {}-{} is {}'.format(i,t[0],t[1],v))
30
31 # The above loop prints the following:
32 # (0, 0) A-A is Lennard-Jones
33 # (0, 1) A-B is Weeks-Chandler-Andersen
34 # (0, 2) A-C is Exponential
35 # (1, 1) B-B is Weeks-Chandler-Andersen
36 # (1, 2) B-C is Weeks-Chandler-Andersen
37 # (2, 2) C-C is Exponential
38
39 for i,t,v in PT.iterpairs(full=True):
40     print('{} {}-{} is {}'.format(i,t[0],t[1],v))
41
42 # The above loop prints the following:
43 # (0, 0) A-A is Lennard-Jones
44 # (0, 1) A-B is Weeks-Chandler-Andersen
45 # (0, 2) A-C is Exponential
46 # (1, 0) B-A is Weeks-Chandler-Andersen
47 # (1, 1) B-B is Weeks-Chandler-Andersen
48 # (1, 2) B-C is Weeks-Chandler-Andersen
49 # (2, 0) C-A is Exponential
50 # (2, 1) C-B is Weeks-Chandler-Andersen
51 # (2, 2) C-C is Exponential
```

In some cases where additional logic or error checking is needed, we have created more specialized container classes. For example, both the site volumes and the site-site contact distances are functions of the individual site diameters. The pyPRISM.Diameter class contains multiple pyPRISM.Table objects which are dynamically updated as the user defines site-type diameters. The pyPRISM.Density class was created for analogous reasons so that the pair-density matrix,

$$\rho_{\alpha,\beta}^{pair} = \rho_{\alpha}\rho_{\beta}$$

the site-density matrix,

$$\rho_{\alpha,\beta}^{site} = \begin{cases} \rho_{\alpha} & \text{if } i = j \\ \rho_{\alpha} + \rho_{\beta} & \text{if } i \neq j \end{cases}$$

and the total site density,

$$\rho^{total} = \sum_{\alpha} \rho_{\alpha,\alpha}^{site}$$

can all be calculated dynamically as the user specifies or modifies the individual site-type densities ρ_α .

An additional specialized container is `pyPRISM.Domain`. This class specifies the discretized real- and Fourier-space grids over which the PRISM equation is solved and is instantiated by specifying the length (i.e., number of gridpoints) and grid spacing in real- or Fourier space (i.e., dr or dk). An important detail of the PRISM cost function mentioned above is that correlation functions need to be transformed to and from Fourier space during the cost function evaluation. `pyPRISM.Domain` also contains the Fast Fourier Transform (FFT) methods needed to efficiently carry out these transforms. The mathematics behind these FFT methods, which are implemented as Type II and III Discrete Sine Transforms (DST-II and DST-III), are discussed in our previous work. [MGIJ⁺18]

The `pyPRISM.System` class contains multiple `pyPRISM.ValueTable` and `pyPRISM.PairTable` objects in addition to the specialized container classes described above. The goal of the `pyPRISM.System` class is to be a super-container that can validate that a system is fully and correctly specified before allowing the user to attempt to solve the PRISM equation.

While `pyPRISM.System` primarily houses input property tables, `pyPRISM.PRISM` represents a fully specified PRISM calculation and contains the cost function to be numerically minimized. The correlation functions shown in Equation 1 are stored in the `pyPRISM.PRISM` object as `pyPRISM.MatrixArray` objects, which are similar to `pyPRISM.ValueTable` objects but with a focus on mathematics rather than storage. `pyPRISM.MatrixArray` objects can only contain numerical data and provide many operators and methods which simplify PRISM theory mathematics. In particular, they satisfy the need for easy access to both the matrix and pair-function representations of the correlation functions, shown schematically in Figure 4. The former is necessary for carrying out the mathematics of the PRISM equation (Equation 1) and the latter for performing Fourier transformations of the individual pair-functions. The `pyPRISM.MatrixArray` objects also carry out a number of run-time error checks including ensuring that both `MatrixArray` objects involved in a binary operations (such as addition) are in the same space (real or Fourier). The core data structure underlying the `pyPRISM.MatrixArray` is a three-dimensional Numpy ndarray of $m \times n \times n$ matrices, where m is the length of the `pyPRISM.Domain`.

```
1 '''
2 Example of MatrixArray usage.
3 '''
4 ## Setup ##
5 length = 1024          # number of gridpoints
6 dr = 0.1              # real-space grid spacing
7 rank = 2              # number of site-types
8 types = ['A', 'B']   # name of site-types
9
10 domain = pyPRISM.Domain(length, dr)
11 rho = pyPRISM.Density(types)
12
13 # Total and intra-molecular correlation functions
14 # dataH and dataW are size (length,rank,rank)
15 # numpy ndarrays that are assumed to be in memory
16 kwargs = dict(length=length,rank=rank,types=types)
17 H = pyPRISM.MatrixArray(data=dataH,**kwargs)
18 W = pyPRISM.MatrixArray(data=dataW,**kwargs)
19
20 ## Example Calculation of Structure Factor ##
21 S = (W + H)/rho.site
```

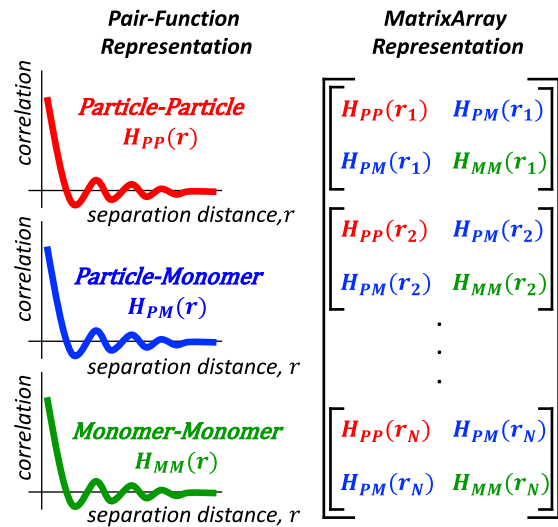


Fig. 4: Schematic of the pair-function and `MatrixArray` representations of the total correlation function for the polymer nanocomposite system shown in Figure 1. The r_1, r_2, r_N variables represent specific distances in the real-space solution grid.

```
22 S_AB = S['A', 'B'] # extract S_AB from MatrixArray
23
24 ## MatrixArray by Scalar Operations ##
25 # All matrices in W are modified by the scalar x
26 x = 1 # arbitrary scalar
27 W+x; W-x; W*x; W/x; # elementwise ops
28
29 ## MatrixArray by Matrix Operations ##
30 # All matrices in W are modified by the matrix rho
31 W+rho; W-rho; W*rho; W/rho; # elementwise ops
32 W.dot(rho) # matrix mult.
33
34 ## MatrixArray by MatrixArray Operations ##
35 # Operations are matrix to corresponding matrix
36 W+H; W-H; W*H; W/H; # elementwise ops
37 W.dot(H) # matrix mult.
38
39 ## Fourier Transformations ##
40 # Transform a single array versus all functions
41 # in a MatrixArray
42 W_AA = domain.to_real(W['A', 'A']) # one function
43 domain.MatrixArray_to_fourier(H) # all functions
44
45 ## Other Operations ##
46 W.invert() # invert each matrix in W
47 W['A', 'B'] # set or get function for pair A-B
48 W.getMatrix(i) # get matrix i in MatrixArray
49 W.iterpairs() # iterate over all 1-D functions
```

The `pyPRISM.PRISM` object is solved by calling the `.solve()` method which invokes a numerical algorithm to minimize the output of the `.cost()` method by varying the input $\Gamma_{\alpha,\beta}(r)$. Once a `pyPRISM.PRISM` object is numerically solved, it can be passed to a calculator that processes the optimized correlation functions and returns various structural and thermodynamic data. The current list of available calculators is shown in the rightmost column of Figure 3 and is fully described in the documentation. [pyPa]

Beyond the core data structures, `pyPRISM` defines classes which are meant to represent various theoretical equations or ideas. Classes which inherit from `pyPRISM.Potential`, `pyPRISM.Closure`, or `pyPRISM.Omega` represent interaction potentials, theoretical closures, or *intra*-molecular correla-

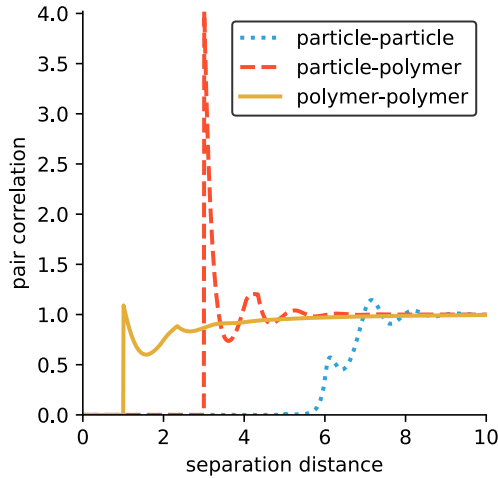


Fig. 5: All pair-correlation functions from the pyPRISM example for the polymer nanocomposite system depicted in Figure 1.

tion functions $\hat{\Omega}_{\alpha,\beta}(k)$, respectively. These properties must be specified for all site-type pairs before a pyPRISM.PRISM object can be created. To ensure that users can easily add new potentials, closures, and $\hat{\Omega}_{\alpha,\beta}(k)$ to the codebase, we have kept the programming interface contract of these classes as simple as possible: Subclasses must inherit from the proper parent class and implement a `.calculate()` method.

Example pyPRISM Script

```

1 '''
2 pyPRISM script calculating the pair correlation
3 functions and chi parameters of a polymer
4 nanocomposite.
5 '''
6
7 import pyPRISM
8
9 sys = pyPRISM.System(['particle', 'polymer'], kT=1.0)
10 sys.domain = pyPRISM.Domain(dr=0.01, length=4096)
11
12 sys.diameter['polymer'] = 1.0
13 sys.diameter['particle'] = 5.0
14
15 sys.density['polymer'] = 0.75
16 sys.density['particle'] = 6e-6
17
18 sys.omega['polymer', 'polymer'] = \
19 pyPRISM.omega.FreelyJointedChain(length=100, l=4/3)
20 sys.omega['polymer', 'particle'] = \
21 pyPRISM.omega.InterMolecular()
22 sys.omega['particle', 'particle'] = \
23 pyPRISM.omega.SingleSite()
24
25 sys.potential['polymer', 'polymer'] = \
26 pyPRISM.potential.HardSphere()
27 sys.potential['polymer', 'particle'] = \
28 pyPRISM.potential.Exponential(alpha=0.5, epsilon=1.0)
29 sys.potential['particle', 'particle'] = \
30 pyPRISM.potential.HardSphere()
31
32 sys.closure['polymer', ['polymer', 'particle']] = \
33 pyPRISM.closure.PercusYevick()
34 sys.closure['particle', 'particle'] = \
35 pyPRISM.closure.HyperNettedChain()
36
37 PRISM = sys.solve()
38
39 pcf = pyPRISM.calculate.pair_correlation(PRISM)

```

```

40 pcf_11 = pcf['particle', 'particle']
41
42 chi = pyPRISM.calculate.chi(PRISM)
43 chi_12 = pcf['particle', 'polymer']

```

Example Discussion

The code above shows how to use pyPRISM to calculate the properties of a polymer nanocomposite made of linear polymer chains and spherical nanoparticles. This system is shown schematically in Figure 1 and is fully described in reference [HS05]. The results of this calculation are plotted in Figure 5. In this section, we will discuss the details of this example in a line by line fashion as we specify all inputs shown in Figure 2 and then solve the PRISM equation.

```
6 import pyPRISM
```

```
7
8 sys = pyPRISM.System(['particle', 'polymer'], kT=1.0)
9 sys.domain = pyPRISM.Domain(length=4096, dr=0.01)
```

All pyPRISM calculations begin by first importing the pyPRISM library, and then creating a pyPRISM.System object. The first argument to the pyPRISM.System constructor is the names of the site-types for the calculation. In this case, we have two site-types which we (arbitrarily) call *polymer* and *particle*. Optionally, the constructor allows that the thermal energy level, $k_B T$, be specified. Next a pyPRISM.Domain object is created with `length=4096` grid-points and a grid spacing of `dr=0.1`.

Note that all parameters in pyPRISM are specified in a reduced unit system commonly called Lennard-Jones units. In this scheme, a characteristic length d_c , mass m_c , and energy e_c are specified. All other units are then specified in terms of these characteristic units. For example, if $d_c = 1$ nm, the grid spacing in the above code would be $dr = 0.1d_c = 0.1$ nm. See [FB02] for more information on the Lennard-Jones reduced unit scheme.

```

11 sys.diameter['polymer'] = 1.0
12 sys.diameter['particle'] = 5.0
13
14 sys.density['polymer'] = 0.75
15 sys.density['particle'] = 6e-6

```

Next, site-type diameters and number densities are specified for both site-types in units of d_c and beads per d_c^3 , respectively. Qualitatively, these specifications imply that we are considering a dilute concentration of nanoparticles dissolved in a polymer matrix made up of polymer sites of significantly smaller diameter.

```

17 sys.omega['polymer', 'polymer'] = \
18 pyPRISM.omega.FreelyJointedChain(length=100, l=4/3)
19 sys.omega['polymer', 'particle'] = \
20 pyPRISM.omega.InterMolecular()
21 sys.omega['particle', 'particle'] = \
22 pyPRISM.omega.SingleSite()

```

The *intra*-molecular correlation function $\hat{\Omega}_{polymer,polymer}(k)$ is specified as a freely jointed chain, a well-known physical model for a polymer chain. [RC03] Since the polymer chains and particles are not connected, $\hat{\Omega}_{polymer,particle}(k)$ is specified as *inter*-molecular. The particles are modeled as spherical sites so $\hat{\Omega}_{particle,particle}(k)$ is modeled as a pyPRISM.omega.SingleSite.

```

24 sys.potential['polymer', 'polymer'] = \
25 pyPRISM.potential.HardSphere()
26 sys.potential['polymer', 'particle'] = \
27 pyPRISM.potential.Exponential(alpha=0.5, epsilon=1.0)
28 sys.potential['particle', 'particle'] = \
29 pyPRISM.potential.HardSphere()

```

$U_{polymer,polymer}(r)$ and $U_{particle,particle}(r)$ pair potentials are specified as athermal hard sphere interactions, while the $U_{polymer,particle}(r)$ potential is an exponential attractive interaction. This configuration describes a dense melt-like polymer nanocomposite where the polymer chains are attracted to and adhere to (wet) the nanoparticle surface. The α and ε parameters in the `pyPRISM.Potential.Exponential` constructor control the range and strength of the exponential attraction.

```
31 sys.closure['polymer', ['polymer', 'particle']] = \
32 pyPRISM.closure.PercusYevick()
33 sys.closure['particle', 'particle'] = \
34 pyPRISM.closure.HyperNettedChain()
```

To demonstrate one utility of the `pyPRISM.PairTable` data structure, here we have specified both the *polymer-polymer* and *polymer-particle* closure in a single line. Both pair-data are specified to the Percus-Yevick closure, while the *particle-particle* closure is set to be the hypernetted chain closure. In this code-block and those above, note how the subclasses of `pyPRISM.Omega`, `pyPRISM.Potential` and `pyPRISM.Closure` are used to easily specify complex theoretical constructs.

```
36 PRISM = sys.solve()
```

When all properties are defined, the user calls the `pyPRISM.System.solve()` method which first conducts a number of sanity checks and issues any relevant exceptions or warnings if issues are found. If no issues are found, a PRISM object is created and minimization is attempted. The `.solve()` method accepts arguments which allow the user to tune the details of the minimization.

```
38 pcf = pyPRISM.calculate.pair_correlation(PRISM)
39 pcf_11 = pcf['particle', 'particle']
40
41 chi = pyPRISM.calculate.chi(PRISM)
42 chi_12 = pcf['particle', 'polymer']
```

Once the minimization completes, a `pyPRISM.PRISM` object is returned which contains the final solutions for $H(r)$ and $C(r)$ along with all input parameters and data. The `pyPRISM.PRISM` object is then passed through the `pyPRISM.calculate.pair_correlation` and `pyPRISM.calculate.chi` calculators. Both of these methods return `pyPRISM.ValueTables`, which can be subscripted to access the individual pair-functions. In the example, we extract the particle-particle pair correlation function, $g_{particle,particle}(r)$ and the particle-polymer $\chi_{particle,polymer}$ parameter.

While it would be feasible to study this polymer nanocomposite system *via* simulation methods such as MD or MC, the use of PRISM theory offers some distinct advantages. PRISM theory does not suffer from finite-size or equilibration effects, both of which limit simulation methods. Furthermore, a simulation of sufficient size to study the large nanoparticles and relatively long polymer chains in this example would require many hours to days of CPU or GPU time from a supercomputing resource. This is due to the computational expense of evaluating the pairwise interactions at each simulated configuration and the many millions of configurations that must be generated in order to properly equilibrate and sample such a nanocomposite. In contrast, PRISM theory can be numerically solved in seconds even on modest hardware such as a laptop computer. This is because, unlike MD or MC, solving PRISM theory does not involve generating molecular configurations, but rather is a set of integral equations which are numerically solved for the spatial correlation functions,

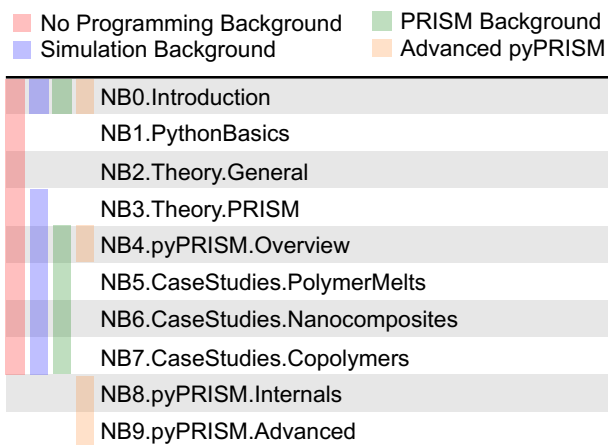


Fig. 6: Depiction of the tutorial tracks we provide for users of different backgrounds and trainings. See the Tutorial page [[pyPb](#)] for more information.

$H_{\alpha,\beta}(r)$ and $C_{\alpha,\beta}(r)$. This numerical solution process is briefly described above at the end of the [PRISM Theory](#) section and is described in detail in Section II.E of [[MGJ⁺18](#)]. In addition to the computational performance benefits of PRISM theory over MD or MC, once the full set of pairwise spatial correlation functions is solved for, a variety of properties can quickly be screened without having to process large simulation trajectories.

PRISM theory provides a powerful alternative or complement to traditional simulation approaches, but we should note that it is not without limitation. There are restrictions on the types of systems and thermodynamic state points to which PRISM theory can be applied and the numerical closures are approximations and therefore sources of error. See Section IV.D of [[MGJ⁺18](#)] for a discussion on the known limitations of PRISM theory.

Pedagogy

It is our goal to create a central platform for polymer liquid state theorists while also lowering the barriers to using PRISM theory for the greater polymer science community. Towards this effort, we have identified two primary challenges:

- 1) The process of understanding and numerically solving PRISM theory is complex and filled with pitfalls and opportunities for error.
- 2) Many of those who would benefit most from PRISM theory do not have a strong programming background.

Our strategy to address both of these challenges is a strong focus on providing pedagogical resources to users. To start, we have put significant effort into our documentation. Every page of the API documentation [[pyPa](#)] contains a written description of the theory being implemented, all necessary mathematics, descriptions of all input and output parameters, links to any relevant journal articles, and a detailed and relevant example. While including these features in our documentation is not a new idea, we are focusing on providing these resources immediately upon release and iterating based on user feedback to improve the clarity and scope of the information provided.

Moving beyond API documentation, we also have created knowledgebase materials which provide more nuanced information about using and numerically solving PRISM theory. This

knowledgebase includes everything from concise lists of systems and properties that can be studied with pyPRISM to tips and tricks for reaching convergence of the numerical solver. In reference to Challenge 2 above, we also recognize that a significant barrier for non-experts to use these tools is the installation process. Our installation documentation [pyPa] attempts to be holistic and provide detailed instructions for the several different ways that users can install pyPRISM.

We have also created a self-guided tutorial to PRISM theory and pyPRISM in the form of a series of Jupyter notebooks. [pyPb], [jup] The tutorial notebooks are designed to target a wide audience with varied programming and materials science expertise, with topics ranging from a basic introduction to Python to how to add new features to pyPRISM. The tutorial also has several case study-focused notebooks which walk users through the process of reproducing PRISM results from the literature. Figure 6 shows our recommendations for how users of different backgrounds and skill levels might move through the tutorial. In order to ensure the widest audience possible can take advantage of this tutorial, we have also set up a binder instance [pyPc], which allows users to try out pyPRISM and run the tutorial instantly in a web-browser without installing any software. This feature should also benefit users who might be hampered by Challenge 2 above.

Future Directions

While pyPRISM is a step forward in providing a central platform for polymer liquid-state theory calculations, we intend to significantly extend the tool beyond its release state. The most obvious avenue for extension will be to add new potentials, closures, and *intra*-molecular correlation functions ($\hat{\Omega}_{\alpha,\beta}(k)$) to the codebase. As described above, we hope that a significant portion of these classes will be contributed by users. Where analytical expressions for $\hat{\Omega}_{\alpha,\beta}(k)$ do not exist, they can also be calculated from simulation trajectories. While we do provide a Cython-enhanced tool to do the calculation, we also plan to add features to more easily couple pyPRISM to common MD and MC simulation packages. [hoo], [lam], [sim], [cas] These linkages would also make it easier for users to carry out the Self-Consistent PRISM (SCPRISM) method. [MGIJ+18]

PRISM theory also has advanced applications that are not possible in the current pyPRISM workflow. One example is the use of PRISM theory to translate a detailed atomistic simulation model to a less detailed, less computationally expensive coarse-grained model in a methodology called Integral Equation Coarse Graining (IECG). [DG17b], [DG17a], [MCLG12], [YSNG04] We plan to provide utilities in the pyPRISM codebase that aid in carrying out this method. PRISM theory can also be used to model or fit neutron and X-ray scattering data. In particular, PRISM theory can be used to take existing scattering models for single particles or polymer chains and model the effects of intermolecular interactions. This approach would greatly extend the applicability of existing scattering models, which on their own are only valid in the infinitely dilute concentration limit, but could be combined with pyPRISM to model higher concentrations.

Summary

pyPRISM is an open-source tool with the goal of facilitating the usage of PRISM theory, a polymer liquid-state theory. Compared to more widely-used simulation methods such as MD and MC, PRISM theory is significantly more computationally efficient,

does not need to be equilibrated, and does not suffer from finite size effects. pyPRISM lowers the barriers to using PRISM theory by providing a simple scripting interface for setting up and numerically solving the theory. Furthermore, in order to ensure users correctly and appropriately use pyPRISM, we have created extensive pedagogical materials in the form of API documentation, knowledgebase materials, and Jupyter-notebook powered tutorials.

Acknowledgements

TBM is supported by a National Research Council (NRC) fellowship at the National Institute of Standards and Technology (NIST). In addition, this work has been supported by the members of the NIST nSoft consortium (nist.gov/nsoft). TEG and AJ thank National Science Foundation Division of Materials Research Condensed Matter and Materials Theory (NSF DMR-CMMT) grant number 1609543 for financial support. This research was supported in part through the use of Information Technologies (IT) resources at the University of Delaware, specifically the high-performance computing resources of the Farber supercomputing cluster. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede cluster at the University of Texas through allocation MCB100140 (AJ), which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

- [ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. doi:10.1016/j.jcp.2008.01.047.
- [AQM+16] Akash Arora, Jian Qin, David C. Morse, Kris T. Delaney, Glenn H. Fredrickson, Frank S. Bates, and Kevin D. Dorfman. Broadly accessible self-consistent field theory for block polymer materials discovery. *Macromolecules*, 49(13):4675–4690, 2016. doi:10.1021/acs.macromol.6b00107.
- [cas] URL: <https://www3.nd.edu/~ed/research/cassandra.html>.
- [cyt] URL: <http://cython.org>.
- [DG17a] M. Dinpajoo and M. G. Guenza. Thermodynamic consistency in the structure-based integral equation coarse-grained method. *Polymer*, 117:282–286, 2017. doi:https://doi.org/10.1016/j.polymer.2017.04.025.
- [DG17b] Mohammadhasan Dinpajoo and Marina G. Guenza. On the density dependence of the integral equation coarse-graining effective potential. *The Journal of Physical Chemistry B*, 2017. doi:10.1021/acs.jpcc.7b10494.
- [dis] Any identification of commercial or open-source software in this paper is done so purely in order to specify the methodology adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the softwares identified are necessarily the best available for the purpose.
- [FB02] Daan Frenkel and Smit Berend. *Monte Carlo Simulations: A Basic Monte Carlo Algorithm*, book section 3, pages 40–42. Computational Science Series. Academic Press, San Diego, California, 2 edition, 2002.
- [GNA+15] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97 – 107, 2015. doi:10.1016/j.cpc.2015.02.028.
- [hoo] URL: <http://glotzerlab.engin.umich.edu/hoomd-blue/index.html>.
- [HS05] Justin B. Hooper and Kenneth S. Schweizer. Contact aggregation, bridging, and steric stabilization in dense polymer-particle mixtures. *Macromolecules*, 38(21):8858–8869, 2005. doi:10.1021/ma051318k.
- [jup] URL: <https://jupyter.org>.
- [lam] URL: <http://lammmps.sandia.gov/>.

- [MCLG12] J. McCarty, A. J. Clark, I. Y. Lyubimov, and M. G. Guenza. Thermodynamic consistency between analytic integral equation theory and coarse-grained molecular dynamics simulations of homopolymer melts. *Macromolecules*, 45(20):8482–8493, 2012. URL: <http://pubs.acs.org/doi/pdfplus/10.1021/ma301502w>, doi:10.1021/ma301502w.
- [MGJ⁺18] T. B. Martin, T. E. Gartner III, R. L. Jones, C. R. Snyder, and A. Jayaraman. pyprism: A computational tool for liquid-state theory calculations of macromolecular materials. *Macromolecules*, 51(8):2906–2922, 2018. doi:10.1021/acs.macromol.8b00011.
- [new] URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton_krylov.html.
- [num] URL: <http://numpy.org/>.
- [Oli07] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. URL: <http://ieeexplore.ieee.org/document/4160250/>, doi:10.1109/MCSE.2007.58.
- [OS05a] F. T. Oyerokun and K. S. Schweizer. Theory of glassy dynamics in conformationally anisotropic polymer systems. *Journal of Chemical Physics*, 123(22), 2005. URL: <http://aip.scitation.org/doi/pdf/10.1063/1.2135776>, doi:10.1063/1.2135776.
- [OS05b] F. T. Oyerokun and K. S. Schweizer. Thermodynamics, orientational order and elasticity of strained liquid crystalline melts and elastomers. *Journal of Physical Chemistry B*, 109(14):6595–6603, 2005. URL: <http://pubs.acs.org/doi/pdfplus/10.1021/jp045646i>, doi:10.1021/jp045646i.
- [pin] URL: <http://pint.readthedocs.io/>.
- [Pli95] S. Plimpton. False parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995. doi:10.1006/jcph.1995.1039.
- [PS99] G. T. Pickett and K. S. Schweizer. Liquid-state theory of anisotropic flexible polymer fluids. *Journal of Chemical Physics*, 110(14):6597–6600, 1999. URL: <http://aip.scitation.org/doi/pdf/10.1063/1.478566>, doi:10.1063/1.478566.
- [PS00] G. T. Pickett and K. S. Schweizer. Liquid crystallinity in flexible and rigid rod polymers. *Journal of Chemical Physics*, 112(10):4881–4892, 2000. URL: <http://aip.scitation.org/doi/pdf/10.1063/1.481039>, doi:10.1063/1.481039.
- [psc] URL: <http://pscf.cems.umn.edu/>.
- [pyPa] URL: <http://pyprism.readthedocs.io/>.
- [pyPb] URL: <http://pyprism.readthedocs.io/en/latest/tutorial/tutorial.html>.
- [pyPc] URL: <https://mybinder.org/v2/gh/usnistgov/pyprism/master?filepath=tutorial>.
- [pyPd] URL: <https://github.com/usnistgov/pyPRISM>.
- [pyPe] URL: <https://anaconda.org/conda-forge/pyprism>.
- [pyPf] URL: <https://pypi.org/project/pyPRISM/>.
- [RC03] M. Rubinstein and R.H. Colby. *Polymer Physics*. OUP Oxford, 2003.
- [RM11] Neeraj Rai and Edward J. Maginn. Vapor–liquid coexistence and critical behavior of ionic liquids via molecular simulations. *The Journal of Physical Chemistry Letters*, 2(12):1439–1443, 2011. doi:10.1021/jz200526z.
- [SC87] K. S. Schweizer and J. G. Curro. Integral-equation theory of the structure of polymer melts. *Physical Review Letters*, 58(3):246–249, 1987. doi:10.1103/PhysRevLett.58.246.
- [SC94] K. S. Schweizer and J. G. Curro. *PRISM Theory of the Structure, Thermodynamics, and Phase-Transitions of Polymer Liquids and Alloys*, volume 116 of *Advances in Polymer Science*, pages 319–377. 1994. doi:10.1007/BFb0080203.
- [sci] URL: <http://scipy.org/>.
- [sim] URL: <http://dmorse.github.io/simpatico/index.html>.
- [WCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. URL: <http://ieeexplore.ieee.org/document/5725236/>, doi:10.1109/MCSE.2011.37.
- [YSNG04] G. Yatsenko, E. J. Sambriski, M. A. Nemirovskaya, and M. Guenza. Analytical soft-core potentials for macromolecular fluids and mixtures. *Physical Review Letters*, 93(25), 2004. doi:10.1103/PhysRevLett.93.257803.

A Bayesian's journey to a better research workflow

Konstantinos Vamvourellis^{‡*}, Marianne Corvellec[§]

<https://youtu.be/piQvcVala9I>

Abstract—This work began when the two authors met at a software development meeting. Konstantinos was building Bayesian models in his research and wanted to learn how to better manage his research process. Marianne was working on data analysis workflows in industry and wanted to learn more about Bayesian statistics. In this paper, the authors present a Bayesian scientific research workflow for statistical analysis. Drawing on a case study in clinical trials, they demonstrate lessons that other scientists, not necessarily Bayesian, could find useful in their own work. Notably, they can be used to improve productivity and reproducibility in any computational research project.

Index Terms—Bayesian statistics, life sciences, clinical trials, probabilistic programming, Stan, PyStan

Introduction

We present a research workflow for Bayesian statistical analysis. We demonstrate lessons we learned from our own computational research that other scientists, not necessarily Bayesian, could find useful when they manage their work. To illustrate these lessons, we use a specific case study in clinical trial modeling.

Clinical trial data are presented to experts and clinicians to assess the efficacy and safety of a given drug. The analysis of trial data is based on statistical summaries of the data, including averages, standard deviations, and significance levels. However, dependencies between the treatment effects are the subject of clinical judgment and are rarely included in the statistical summaries.

We propose a Bayesian approach to model clinical trial data. We use latent variables to account for the whole joint distribution of the treatment effects, including effects of different types. As a result, we can find the predictive distribution of the treatment effects on a new patient accounting for uncertainty in all the parameters, including correlation between the effects.

The analysis is implemented in PyStan, the Python interface to Stan, which is the state-of-the-art, free and open-source Bayesian inference engine. Stan and the researchers behind it provide users with guidance that make Bayesian inference easier to use. We discuss aspects of this ecosystem in the second-to-last section.

Although this case study is by no means the ideal introductory example of computational modeling, it provides us with a real-world problem from which we can share practical lessons. We believe this paper can be of help to a number of different audiences. Firstly, it can help non-Bayesian statisticians, or beginning

Bayesians, get a sense of how to apply Bayesian statistics to their work. Secondly, it can provide computational scientists with advice on building a reproducible and efficient research workflow. And, thirdly, it can spark discussions among advanced Bayesians about the complexities of Bayesian workflows and how to build better models.

A Bayesian Workflow

We present a Bayesian workflow for statistical modeling. We recognize that the research process is too complex to summarize in a recipe-style list. However, we find that there are a few building blocks that are common to every Bayesian statistical analysis. In this paper, we focus on these and break them down to a few basic steps. To avoid over-simplification, we hint to possible connections with more advanced aspects where appropriate. We believe that following a workflow, such as suggested here, can help researchers avoid mistakes and increase productivity. It also helps make research projects more reproducible, as we discuss in the last section.

We propose a simple workflow made of the following steps:

- 1) Scope the problem;
- 2) Specify the likelihood and priors;
- 3) Generate synthetic data that resemble the true data to a reasonable degree;
- 4) Fit the model to the synthetic data;
 - a. Check that the true values are recovered;
 - b. Check the model fit;
- 5) Fit the model to the real data.

An advanced workflow, which is beyond the scope of this paper, could be extended to include the following steps:

- 6) Check the predictive accuracy of the model;
- 7) Evaluate the model fit;
- 8) Select best model among different candidates (model selection);
- 9) Perform a sensitivity analysis.

In what follows, we will use $M(\theta)$ to denote the model as a function of its parameter θ (θ is either a scalar or a vector representing a set of parameters). Data usually consist of observable outcomes¹ y and covariates² x , if any. We will distinguish between the two when necessary; otherwise, we will denote all data together by D . We use $p(\cdot)$ to denote either probability

* Corresponding author: k.vamvourellis@lse.ac.uk

‡ London School of Economics and Political Science

§ Institute for Globally Distributed Open Research and Education (IGDORE)

distributions or probability densities, even though it is not rigorous notation.

1) Scope the problem

The main goal of this workflow is to achieve successful Bayesian inference. That is, correctly retrieving samples from the posterior distribution of the parameter values, which are typically unknown before the analysis, using the information contained in the data. The major difference of the Bayesian approach relative to frequentist, is that it modifies the likelihood function (to be introduced later) into a proper distribution over the parameters, called the posterior distribution. The posterior distribution $p(\theta|D)$ forms the basis of the Bayesian approach from which we derive all quantities of interest.

Why do we need statistical inference in the first place? We need it to answer our questions about the world. Usually, our questions refer to an implicit or explicit parameter θ in a statistical model, such as:

- What values of θ are most consistent with the data?
- Do the data support a certain condition (e.g., for θ a scalar, $\theta > 0$)?
- How can we predict the future outcome of an experiment?

To proceed, we need to define a model. Choosing a model is usually tied to the exact research questions we are interested in. We can choose to start with a postulated data generation process and then decide how to interpret the parameters in relation to the research question. Alternatively, it is equally valid to start from the research question and design the model so that its parameters are directly connected to the specific questions we wish to answer. In the next section, we illustrate with an example how to design a model to answer a specific research question.

Note that the question of prediction depends directly on inferring successfully the parameter values. We shall come back to this at the end of this section.

2) Specify the likelihood and priors

Once we have defined the scope of the problem, we need to specify the design of the model which is captured in the *likelihood* function $f(D|\theta, M)$. Usually, argument M is dropped for notational simplicity, the model being chosen and assumed known³. Note, however, that when the model includes covariates, the more accurate expression is $f(y|\theta, x)$. This function ties together the ingredients of statistical inference and allows information to flow from the data D to the parameters θ . With Bayes' rule, $p(\theta|D) = p(D|\theta)p(\theta)/p(D)$, we can calculate the posterior distribution.

The second ingredient of Bayesian inference is the prior distribution $p(\theta)$. Priors are inescapably part of the Bayesian approach and, hence, have to be considered carefully. The goal of Bayesian inference is to combine the prior information on the parameters (the prior distribution), with the evidence contained in the data (the likelihood), to derive the posterior distribution $p(\theta|D)$. It is difficult to predict how sensitive the final results will be to a change in the priors. However, it is important to note that

1. Depending on their field, readers may want to think 'dependent variables' or 'labels'.

2. Depending on their field, readers may want to think 'independent variables' or 'features'.

3. This is a good time to highlight that the choice of the model is a constant assumption in everything we do from now on. In research projects, it is common to work with a few different models in parallel.

the impact of priors progressively diminishes as the number of observations increases.

The ideal scenario for applying the Bayesian approach is when prior knowledge is available, in which case the prior distribution can and should capture that knowledge. But, sometimes, we might want to avoid expressing prior knowledge, especially when such knowledge is not available. How are we supposed to choose priors then? Constructing default priors is an active area of research that is beyond the scope of this work. Here, we provide a high-level overview and refer the interested reader to various sources for further reading.

Priors which express very little or no prior knowledge are called vague or *uninformative priors*. Such priors are deliberately constructed in a way which minimizes their impact on the resulting inference, relative to the information brought in by the likelihood. In fact, Bayesian inference technically works even when the prior is not a proper distribution but a function that assumes all values are equally likely, referred to as *improper prior*. However, it is generally advisable to avoid improper priors, especially in settings beyond just inference, such as the more advanced workflow of steps 6)–9). If no prior knowledge is available, a normal distribution with large variance is still a better default prior than a uniform distribution. It is important to note that improper or even vague priors are not appropriate for model selection.

Additional considerations can impact the choice of priors, especially when chosen together with the likelihood. From a computational perspective, the most convenient priors are called *conjugate priors*, because they mimic the structure of the likelihood function and lead to a closed-form posterior distribution. Priors can have additional benefits when used with a certain goal in mind. For example, priors can be used to guard against overfitting by pulling the parameters away from improbable values, or help with feature selection (e.g., see horse-shoe priors).

Bayesian critics often see priors as a weakness, whereas in reality they are an opportunity. Notably, priors give us the opportunity to employ our knowledge to guide the inference in the absence of evidence from the data. Also, it is important to remember that, in a scientific research context, we rarely have absolutely no prior knowledge and we typically do not consider any parameter value to be equally likely.

3) Generate synthetic data

Once we have agreed on a generative process, i.e., a model M , we can use it to simulate data D' . To do that, we choose reasonable parameter values θ_0 and use M to generate data based on these values. Alternatively, instead of coming up with reasonable parameter values, we can sample these values from the prior distribution $\theta_0 \sim p(\theta)$. The synthetic data D' can then be interpreted as our prior distribution of the data. Hence, by inspecting the synthetic data, we can reflect back on our choices for the likelihood and priors. However, if we do use our priors to generate parameter values, we should make sure that our priors are not uninformative, which would likely produce unreasonable synthetic data.

Note how the model M is a hypothesized process and comes with necessary assumptions and simplifications. It is highly unlikely that the real world would follow exactly M . That being said, if M is close enough to the real generative process, it can still be very useful to help us understand something about the world. As the phrase goes, "all models are wrong, but some models are useful."

4) Fit the model to the synthetic data

If simulating data using our generative process M is the forward direction, statistical inference is the reverse direction by which we find what parameter values could have produced such data, under M .

The most popular statistical inference algorithm is maximum likelihood estimation (MLE), which finds the parameter values that maximize the likelihood given the observed data. To reiterate, under the Bayesian approach, we treat the parameters θ as random variables and express our prior knowledge about θ with the prior probability distribution $p(\theta)$. Bayesian inference is the process of updating our beliefs about θ in light of the data D . The updating process uses Bayes' rule and results in the conditional distribution $p(\theta|D)$, the posterior distribution. Bayesian inference is generally a hard problem. In most cases, we cannot derive the mathematical form of the posterior distribution; instead, we settle for an algorithm that returns samples from the posterior distribution.

When we fit the model to synthetic data, we want to check two things: the correctness of the inference algorithm and the quality of our model.

a. Much like in software testing, we want to check if the inference process works by starting simple and advance progressively to the real challenge. By fitting the model to synthetic data generated from the same model, we effectively rule out issues of mismatch between our model and the real data. Testing the inference algorithm under these ideal conditions allows us to perfect the inference algorithm in a controlled environment, before trying it on the real data. In our experience, this step brings to the surface many bugs in the code as well as issues about the model in general. It offers an added benefit, later on, when we critique the fit of our model M to the real data D . Having confidence in the correctness of our inference process allows us to attribute any mismatch issues to the choice of the model, as opposed to the inference algorithm.

By fitting the model to synthetic data, we recover samples from the posterior distribution of the model parameters. There are various model fit tests to choose from. At a minimum, we need to check that the true parameter values θ_0 are within the range implied by the posterior distributions⁴. Success at this stage is not a sufficient guarantee that the model will fit well to the real data, but it is a necessary condition for proceeding further.

b. Fitting the model to synthetic data is the first opportunity to critique the model M and, if necessary, calibrate it to better suit our needs. This is a good time to catch any issues that affect the quality of the model irrespective of how well it captures reality. For example, an issue that comes up often is non-identifiability, the situation where the likelihood and the data is specified in a way such that there is not enough information to identify the correct parameter values, no matter how big the sample size is. It is also a good time to check if small variations to the model (such as replacing a normal with a heavier-tail distribution) fit our needs better. For instance, calibrating a model to make inferences about the center of a distribution, such as the mean, is relatively easy. On the other hand, we might need to do more extensive calibration if we are interested in the tail behavior of the distribution, such as

4. A common test is to construct an interval that includes 95% of the most likely values, called highest posterior density interval, and check that it covers the true parameter values θ_0 that were used to generate the synthetic data. We should tolerate a few misses, since 95% intervals will not cover the true values 5% of the time, even if the algorithm is perfectly calibrated.

maximum values. If we do choose to use a different model M' , we need to go back to step 2) and start again.

Model evaluation is an essential part of a good workflow. It is a complex task that can be used with both synthetic and real data, providing possibly different insights each time. We do not have space to go into more details in this paper but we provide pointers in the further reading section.

5. Fit the model to the real data

This is the moment we have been waiting for: We are ready to fit our model to the real data and get the final results. Usually, we focus our attention on a specific quantity of interest that is derived from the posterior samples (see further reading for pointers). If we are satisfied with the fit of the model, we are done. In most cases, though, at this stage we are expected to evaluate the model again, this time focusing on how well it captures reality. This step is highly application-specific and requires a combination of statistical expertise and subject-matter expertise (we refer the interested reader to sources later). We note that it is important to build confidence in the power of our inference algorithm before proceeding on to interpreting results. This helps us separate, to the extent possible, inference issues from model issues. At this stage, it is likely that we will come up with a slightly updated model M' . We then have to go back and start again from the beginning.

Posterior Predictive Checks and Model Evaluation

In this subsection, we would like to touch briefly on two topics for more advanced workflows, prediction and model evaluation. The Bayesian posterior predictive distribution is given by the following formula:

$$\begin{aligned} p(\bar{y}|D) &= \int p(\bar{y}, \theta|D) d\theta \\ &= \int p(\bar{y}|\theta) p(\theta|D) d\theta \end{aligned}$$

In practice, we approximate the integral using samples from the posterior distributions. Posterior predictive checks, evaluating the predictive accuracy of a model, can also be used to evaluate a model. To do this, we check how well it predicts unknown observable data \bar{y} , where unknown means that the model was not fit to \bar{y} ⁵.

Further reading

For a concise overview of statistical modeling and inference, including a high-level comparison with the frequentist approach, see [Woo15]. For a more extended treatment of the Bayesian approach, see [Rob07]. For an accessible Bayesian modeling primer, especially for beginner Bayesians, see [McE15] and [MR06]. For a complete treatment of Bayesian data analysis, including many workflow-related discussions, see [GCS+13]⁶.

A Case Study in Clinical Trial Data Analysis

We propose a Bayesian model to extract insights from clinical trial datasets. We are interested in understanding the effect of a treatment on the patients. Our goal is to use the data to predict the effect of the treatment on a new patient. We apply our method on artificially created data, for illustration purposes only.

5. To check the predictive accuracy of the model, we need to measure our predictions \bar{y} against real data. To do this, we usually hold out a small random sample of the original data and deliberately restrain from fitting the model to that sample.

6. And for an example implementation of a complete workflow with PyStan, see https://github.com/betanalphabet/jupyter_case_studies/tree/master/pystan_workflow.

Subject ID	Group Type	Hemoglobin Level	Dyspepsia	Nausea
123	Control	3.42	1	0
213	Treatment	4.41	1	0
431	Control	1.12	0	0
224	Control	-0.11	1	0
233	Treatment	2.42	1	1

TABLE 1: Toy clinical trial data.

1) Scope the problem

Regulators focus on a few key effects when deciding whether a drug is fit for market. In our case we will assume, for simplicity, that there are three effects, where two are binary variables and the other is a continuous variable.

Our dataset is organized as a table, with one patient (subject) per row and one effect per column. For example, if our clinical trial dataset records three effects per subject, ‘Hemoglobin Levels’ (continuous), ‘Nausea’ (yes/no), and ‘Dyspepsia’ (yes/no), the dataset looks like Table 1.

The fact that the effects are of mixed data types, binary and continuous, makes it harder to model their interdependencies. To address this challenge, we use a latent variable structure. Then, the expected value of the latent variables will correspond to the average effect of the treatment. Similarly, the correlations between the latent variables will correspond to the correlations between effects. Knowing the distribution of the latent variables will give us a way to predict what the effect on a new patient will be, conditioned on the observed data.

2) Specify the model, likelihood, and priors

a. Model: Let Y be a $N \times K$ matrix where each column represents an effect and each row refers to an individual subject. This matrix contains our observations, it is our clinical trial dataset. We distinguish between treatment and control subjects by considering separately Y^T (resp. Y^C), the subset of Y containing only treatment (resp. control) subjects. Since the model for Y^T and Y^C is identical, for convenience, we suppress the notation into Y in the remainder of this section.

We consider the following general latent variable framework. We assume subjects are independent and wish to model the dependencies between the effects. The idea is to bring all columns to a common scale $(-\infty, \infty)$. The continuous effects are observed directly and are already on this scale. For the binary effects, we apply appropriate transformations on their parameters via user-specified link functions $h_j(\cdot)$, in order to bring them to the $(-\infty, \infty)$ scale. Let us consider the i -th subject. Then, if the j -th effect is measured on the binary scale, the model is

$$\begin{aligned} Y_{ij} &\sim \text{Bernoulli}(\eta_{ij}) \\ h_j(\eta_{ij}) &= Z_{ij}, \end{aligned}$$

where the link function $h_j(\cdot)$ can be the logit, probit, or any other bijection from $[0, 1]$ to the real line. Continuous data are assumed to be observed directly and accurately (without measurement error), and modeled as follows:

$$Y_{ij} = Z_{ij} \quad \text{for } i = 1, \dots, N.$$

In order to complete the model, we need to define the $N \times K$ matrix Z . Here, we use a K -variate normal distribution $N_K(\cdot)$ on each Z_i .

row, such that

$$Z_i \sim N_K(\mu, \Sigma),$$

where Σ is a $K \times K$ covariance matrix, μ is a row K -dimensional vector, and Z_i are independent for all i .

In the model above, the vector $\mu = (\mu_1, \dots, \mu_K)$ represents the average treatment effect in the common scale. In our example, the first effect (Hemoglobin Level) is continuous and hence its latent value directly observed. Regarding the remaining two effects (Dyspepsia and Nausea), their latent values can only be inferred via their binary observations. Note that the variance of the non-observed latent variables is non-identifiable [CG98], [TDM12], so we need to fix it to a known constant (here we use 1) to fully specify the model. We do this by decomposing the covariance into correlation and variance: $\Sigma = DRD$, where R is the correlation matrix and D is a diagonal matrix of variances $D_{jj} = \sigma_j^2$ for the j -th effect.

b. Likelihood: The likelihood function can be expressed as

$$\begin{aligned} f(Y|Z, \mu, \Sigma) &= f(Y|Z) \cdot p(Z|\mu, \Sigma) \\ &= \left[\prod_{j \in J_b} \prod_{i=1}^N h_j^{-1}(Z_{ij})^{Y_{ij}} (1 - h_j^{-1}(Z_{ij}))^{(1-Y_{ij})} \right] \cdot p(Z|\mu, \Sigma) \\ &= \left[\prod_{j \in J_b} \prod_{i=1}^N \eta_{ij}^{Y_{ij}} (1 - \eta_{ij})^{(1-Y_{ij})} \right] \cdot N(Z|\mu, \Sigma), \end{aligned}$$

where J_b is the index of effects that are binary and $N(Z|\mu, \Sigma)$ is the probability density function (pdf) of the multivariate normal distribution.

c. Priors: In this case study, the priors should come from previous studies of the treatment in question or from clinical judgment. If there was no such option, then it would be up to us to decide on an appropriate prior. We use the following priors for demonstration purposes:

$$\begin{aligned} \mu_i &\sim N(0, 10) \\ R &\sim \text{LKJ}(2) \\ \sigma_j &\sim \text{Cauchy}(0, 2) \text{ for } j \notin J_b \\ Z_{ij} &\sim N(0, 1) \text{ for } j \in J_b. \end{aligned}$$

This will become more transparent in the next section, when we come back to the choice of priors⁷. Let us note that our data contain a lot of information, so the final outcome will be relatively insensitive to the priors.

3) Generate synthetic data

To generate synthetic data, given some values for the parameters (μ, Σ) we only need to follow the recipe given by the model. To fix the parameter values we could sample from the priors we chose, or just choose some reasonable values. Here we picked $\mu = (0.3, 0.5, 0.7)$, $\sigma = (1.3, 1, 1)$, and $R(1, 2) = -0.5$, $R(1, 3) = -0.3$, $R(2, 3) = 0.7$. Then, as the model dictates, we use these values to generate samples of underlying latent variables $Z_i \sim N(\mu, \Sigma)$ ⁸. Each Z_i corresponds to a subject, here we choose to generate 200 subjects. The observed synthetic data Y_{ij} are defined to be equal to Z_{ij} for the effects that are continuous. For

⁷ On the LKJ distribution, see <https://www.sciencedirect.com/science/article/pii/S0047259X09000876>.

the binary effects, we sample Bernoulli variables with probability equal to the inverse logit of the corresponding Z_{ij} value.

Recall that a Bayesian model with proper informative priors, such as the ones we use in this model, can also be used directly to sample synthetic data. As explained in the previous section, we can sample all the parameters according to the prior distributions. The synthetic data can then be interpreted as our prior distribution on the data.

4) Fit the model to the synthetic data

The Stan program encoding this model is the following:

```

1 data {
2   int<lower=0> N;
3   int<lower=0> K;
4   int<lower=0> Kb;
5   int<lower=0> Kc;
6   int<lower=0, upper=1> yb[N, Kb];
7   vector[Kc] yc[N];
8 }
9
10 transformed data {
11   matrix[Kc, Kc] I = diag_matrix(rep_vector(1, Kc));
12 }
13
14 parameters {
15   vector[Kb] zb[N];
16   cholesky_factor_corr[K] L_R;
17   vector<lower=0>[Kc] sigma;
18   vector[K] mu;
19 }
20
21 transformed parameters {
22   matrix[N, Kb] z;
23   vector[Kc] mu_c = head(mu, Kc);
24   vector[Kb] mu_b = tail(mu, Kb); {
25     matrix[Kc, Kc] L_inv = \
26     mdivide_left_tri_low(diag_pre_multiply(sigma, \
27     L_R[1:Kc, 1:Kc]), I);
28     for (n in 1:N) {
29       vector[Kc] resid = L_inv * (yc[n] - mu_c);
30       z[n,] = transpose(mu_b + tail(L_R * \
31       append_row(resid, zb[n]), Kb));
32     }
33 }
34 }
35
36 model {
37   mu ~ normal(0, 10);
38   L_R ~ lkj_corr_cholesky(2);
39   sigma ~ cauchy(0, 2.5);
40   yc ~ multi_normal_cholesky(mu_c, \
41   diag_pre_multiply(sigma, L_R[1:Kc, 1:Kc]));
42   for (n in 1:N) zb[n] ~ normal(0, 1);
43   for (k in 1:Kb) yb[, k] ~ bernoulli_logit(z[, k]);
44 }
45
46 generated quantities {
47   matrix[K, K] R = \
48   multiply_lower_tri_self_transpose(L_R);
49   vector[K] full_sigma = append_row(sigma, \
50   rep_vector(1, Kb));
51   matrix[K, K] Sigma = \
52   multiply_lower_tri_self_transpose(\
53   diag_pre_multiply(full_sigma, L_R));
54 }

```

Model Fit Checks

Figures 1, 2, and 3, we plot the posterior samples on top of the true values (vertical black lines). We check visually that the intervals containing 95% of samples (around their respective means) cover the true values we used to generate the synthetic data.

8. Both $Z_i \sim N_K(\mu, \Sigma)$ and $Z_i \sim N(\mu, \Sigma)$ hold, since the \sim symbol means “is distributed as” and $N(\mu, \Sigma)$ is the pdf of $N_K(\mu, \Sigma)$.

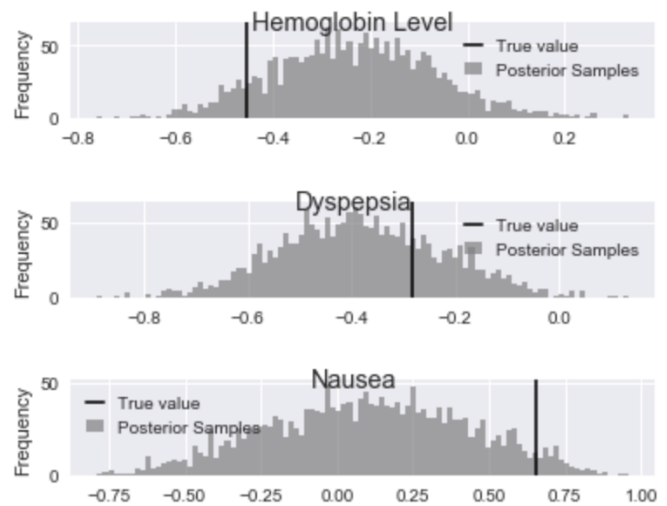


Fig. 1: Histogram of values sampled from the posterior mean of latent variables.

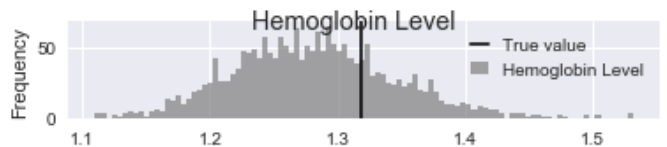


Fig. 2: Histogram of values sampled from the posterior standard deviation for Hemoglobin Level.

With Stan, we can also utilize the built-in checks to inspect the correctness of the inference results. One of the basic tests is the \hat{R} (Rhat), which is a general summary of the convergence of the Hamiltonian Monte Carlo (HMC) chains. Another measure is the number of effective samples, denoted by n_{eff} . Below, we show an excerpt from Stan’s summary of the fit object, displaying Rhat and n_{eff} , along with other metrics (mean and standard deviation), for various parameters. We shall come back to the topic of fit diagnostics in the next section.

Inference for Stan model:

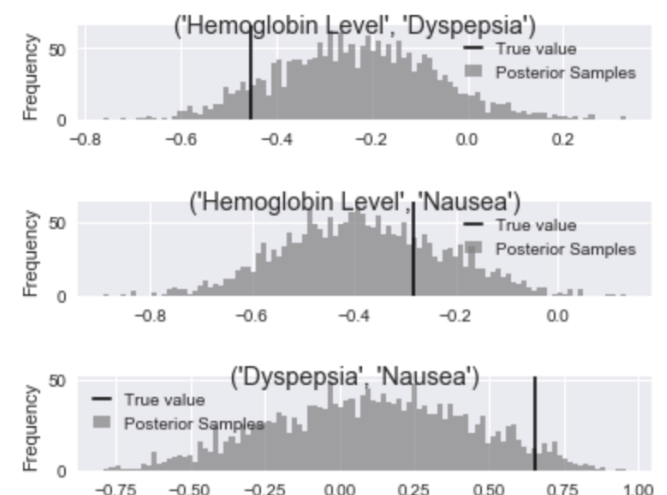


Fig. 3: Histogram of values sampled from the posterior correlation of effects.

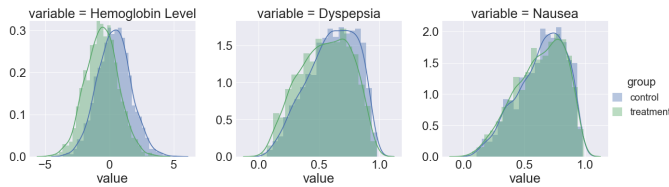


Fig. 4: Histogram of values sampled from the posterior predictive distributions.

```
anon_model_389cd056347577840573e8f6df0e7636.
4 chains, each with iter=1000; warmup=500; thin=1;
post-warmup draws per chain=500,
total post-warmup draws=2000.
```

	mean	sd	...	n_eff	Rhat
mu[0]	0.36	0.09	...	2000	1.0
mu[1]	0.56	0.18	...	2000	1.0
mu[2]	0.67	0.18	...	2000	1.0
R[0,0]	1.0	0.0	...	2000	nan
R[1,0]	-0.24	0.16	...	2000	1.0
R[2,0]	-0.38	0.16	...	2000	1.0
R[0,1]	-0.24	0.16	...	2000	1.0
R[1,1]	1.0	9.3e-17	...	1958	nan
R[2,1]	0.1	0.32	...	550	1.0
R[0,2]	-0.38	0.16	...	2000	1.0
R[1,2]	0.1	0.32	...	550	1.0
R[2,2]	1.0	7.8e-17	...	2000	nan
sigma[0]	1.28	0.06	...	2000	1.0

5. Fit the model to the real data

Once we have built confidence in our inference algorithm, we are ready to fit our model to the real data and answer the question of interest. Our goal is to use the data to predict the effect of the treatment on a new patient, i.e., the posterior predictive distribution.

In this case study, we may not share real data but, for demonstration purposes, we created two other sets of synthetic data, one representing the control group and the other the treatment group. For each posterior sample of parameters (μ_i, Σ_i) , we generate a latent variable $Z_i \sim N(\mu_i, \Sigma_i)$. We then set $Y_{ij} = Z_{ij}$ for $j = 1$, whereas for $j = \{2, 3\}$, we sample $Y_{ij} \sim \text{Bernoulli}(\text{logit}^{-1}(Z_{ij}))$. The resulting set of Y_i is the posterior predictive distribution. We do this for the parameters learned from both groups, Y^T and Y^C separately, and plot the results in Figure 4.

Looking at the plots, we can visualize the effect of the drug on a new patient by distinguishing the effects with the treatment (green) versus without (blue). We observe that the Hemoglobin levels are likely to decrease under the treatment by about 1 unit on average. The probability of experiencing dyspepsia is slightly lower under the treatment, contrary to that of nausea which is the same in both groups. Note how the Bayesian approach results in predictive distributions rather than point estimates, by incorporating the uncertainty from the inference of the parameters.

Bayesian Inference with Stan

Stan is a powerful tool which “mitigates the challenges of programming and tuning” HMC to do statistical inference. Stan is a compiled language written in C++. It includes various useful tools and integrations which make the researcher’s life easier. It can be accessed from different languages via interfaces. This case study was created with the Python interface, Pystan. Note that, at the time of writing, the most developed interfaced is the R one, called RStan. Although the underlying algorithm and speed is the same

throughout the different interfaces, differences in user experience can be meaningful.

Stan requires a description of the basic ingredients of Bayesian inference (i.e., the model, likelihood, priors, and data) and returns samples from the posterior distribution of the parameters. The user specifies these ingredients in separate code blocks called *model* (lines 37–45), *parameters* (lines 14–20), and *data* (lines 1–8). Stan code is passed in via a character string or a plain-text *.stan* file, which is compiled down to C++ when the computation happens. Results are returned to the interface as objects.

Choice of priors

Stan provides many distributions to choose from, which are pre-implemented to maximize efficiency. The Stan team also provides researchers with recommendations on default priors for commonly used parameters, via the Stan manual [Tea17] and other online materials. In our case study, we chose an LKJ prior (line 39) for the correlation matrix, one of the pre-implemented distributions in Stan. The LKJ prior has certain attractive properties and is a recommended prior for correlation matrices in Stan (for reasons beyond the scope of this paper). It has only one parameter (we set it to 2) which pulls slightly the correlation terms towards 0. Another example is the half-Cauchy prior distribution for scale parameters such as standard deviation (line 40). Half-Cauchy is the recommended prior for standard deviation parameters because its support is the positive real line but it has higher dispersion than other alternatives such as the normal distribution. Note that it is easy to truncate any pre-implemented distribution. Stan accepts restrictions on parameters. For example, we restrict the parameter for standard deviation to be positive (line 18). This restriction is then respected when combined with the prior distribution defined later (line 40) to yield a constrained half-Cauchy prior.

Fit diagnostics

HMC has many parameters that need to be tuned and can have a big impact on the quality of the inference. Stan provides many automated fit diagnostics as well as options to tune manually the algorithm, if the default values do not work. For example, the Gelman–Rubin convergence statistic, \hat{R} , comes for free with a Stan fit; effective sample size is another good way to evaluate the fit. In most cases, \hat{R} values need to be very close to 1.0 (± 0.01) for the results of the inference to be trusted, although this on its own does not guarantee a good fit. More advanced topics, such as divergent transitions, step sizes and tree depths are examined in the Stan manual, together with recommendations on how to use them.

Challenges

Stan, and HMC in general, is not perfect and can be challenged in various ways. For example multimodal posterior distribution, which are common in mixture models, are hard to explore⁹.

Another common issue is that mathematically equivalent parameterizations of a model can have vastly different performance in terms of sampling efficiency¹⁰. Although finding the right model parameterization does not admit a simple recipe, the Stan manual [Tea17] provides recommendations to common problems. For example, we can usually improve the sampling performance for normally distributed parameters of the form $x \sim N(\mu, \sigma^2)$ if we use the non-center parameterization $x = \mu + \sigma z$ for $z \sim N(0, 1)$. In our case study, we use this trick, or rather its multivariate version, by targeting the non-centered parts of the latent variable Z (lines 15, 23, 31–32 and 43). Another cause of bad inference results in regression models is correlation among covariates. The way to improve the sampling efficiency of a regression model is to parameterize it using the QR decomposition¹¹. We note that these

issues, among others, that a researcher will encounter when using Stan stem from the difficulties of Bayesian inference, and HMC in particular [BG13], not Stan. The biggest limitation of HMC is that it only works for continuous parameters. As a result we cannot use Stan, or HMC for that matter, to do inference on discrete unknown model parameters. However, in some cases we are able to circumvent this issue¹².

Stan vs PyMC3

In this subsection, we provide a brief overview of the similarities and differences between PyStan and PyMC3, which is another state-of-the-art FLOSS¹³ implementation of automatic Bayesian inference in Python. By ‘automatic,’ we mean that the user only needs to specify the model and the data and the software takes care of the Bayesian inference. Both PyStan and PyMC3 let users fit highly complex Bayesian models, by using HMC under the hood.

Stan and PyMC3 are the same insofar as they serve exactly the same purpose. They both are expressive languages and allow flexible model specification in code. PyMC3 leverages Theano to implement automatic differentiation, whereas Stan relies on its own algorithm. Practitioners report that PyMC3 is easier to get started with (hence, more suitable for prototyping), while Stan is more robust (hence, more suitable for production). For example, Prophet¹⁴ is a timeseries forecasting package by Facebook implemented with Stan. Indeed, there is a rich ecosystem of packages built on top of Stan. However, most of these are available in R only. Most of RStan derived packages follow pre-existing conventions to ease the transition of researchers who want to try Bayesian modeling seamlessly. For example, R users are usually familiar with the *glm* building block for fitting generalized linear models; with the *brms* package¹⁵ users can insert a Bayesian estimates in place of frequentist estimates with minimal changes to their scripts. This way users can easily compare the estimates of the two methods and judge whether the Bayesian approach works for them.

Such packages can also be of use to more advanced users of Bayesian inference as they typically implement the state-of-the-art modeling choices such as default priors and expose the generated Stan code to the user. Hence, interested researchers can learn by essentially using them to generate a baseline Stan code that they can tweak further according to their needs. At the time of writing, PyStan users cannot directly benefit from the Stan ecosystem of packages without leaving Python, at least briefly, as most of the packages above are not available in Python. As a result, we think that PyMC3 seems to be a more complete solution from a Python perspective. PyMC3 is native to Python and hence more integrated into Python than PyStan. PyMC3 also offers more integrated plotting capabilities than PyStan¹⁶.

The value of Stan, in the authors’ view, should be considered beyond the mere software implementation of HMC. Stan consists of a dynamic research community that aims at making Bayesian

inference more accessible and robust. This is achieved through open discussion of all Bayesian topics, many of which are areas of active research. Interested users can learn more about Bayesian inference in general, not just Stan, by reading online and participating in the discussion (see next subsection).

Further reading

The Stan manual [Tea17] is a comprehensive guide to Stan but also includes guidance for Bayesian data analysis in general. For a concise discussion on the history of Bayesian inference programs and the advantages of HMC, see [McE17]. For examples of other case studies and tutorials in Stan, see <http://mc-stan.org/users/documentation/>. For active discussions and advice on how to use Stan, see the Stan forum at <http://discourse.mc-stan.org/>.

Reproducibility

In this last section, we report on our experience of making the case study more reproducible. We consider the definition of reproducibility put forward by [KTD18]. Namely, reproducibility is “the ability of a researcher to duplicate the results of a prior study using the same materials as were used by the original investigator” [RMS18]. To achieve it, we follow the guidance of the three key practices of computational reproducibility [Kit18]:

- 1) Organizing the project into meaningful files and folders;
- 2) Documenting each processing step;
- 3) Chaining these steps together (into a processing *pipeline*).

We care about reproducibility for both high-level and low-level reasons. In the big picture, we want to make the work more shareable, reliable, and auditable. In the day-to-day, we want to save time, catch mistakes, and ease collaboration. We are experiencing these benefits already, having taken a few steps towards computational reproducibility. Finally, let us borrow a quote which is well-known in the reproducible research communities: “Your most important collaborator is your future self.”

The case study presented earlier was not originally set up according to the three practices outlined above. Notably, it used to live in a variety of files (scripts, notebooks, figures, etc.) with no particular structure. File organization is a common source of confusion and frustration in academic research projects. So, the first step we took was to create a clear, relatively standardized directory structure. We went for the following:

```
|-- mixed-data/      <- Root (top-most) directory
                       for the project.
|-- README.md       <- General information about
                       the project.
|-- environment.yml <- Spec. file for reproducing
                       the computing environment.
|-- data/
|  |-- raw/          <- The original, immutable
                       data dump.
|  |-- interim/     <- Intermediate outputs.
|-- models/
```

16. For additional sources on PyMC3 vs Stan comparisons, see:

- https://github.com/jonsedar/pymc3_vs_pystan
- <http://discourse.mc-stan.org/t/jonathan-sedar-hierarchical-bayesian-modelling-with-pymc3-and-pystan/3207>
- <http://andrewgelman.com/2017/05/31/compare-stan-pymc3-edward-hello-world/>
- <https://towardsdatascience.com/stan-vs-pymc3-vs-edward-1d45c5d6da77>
- <https://pydata.org/london2016/schedule/presentation/30/>
- https://github.com/jonsedar/pymc3_vs_pystan

9. See https://github.com/betalpha/knitr_case_studies/tree/master/identifying_mixture_models.

10. See <http://mc-stan.org/users/documentation/case-studies/mle-params.html>.

11. See http://mc-stan.org/users/documentation/case-studies/qr_regression.html.

12. See <http://eleanth.org/blog/2018/01/29/algebra-and-missingness/>.

13. FLOSS stands for “Free/Libre and Open Source Software.”

14. See <https://research.fb.com/prophet-forecasting-at-scale/>.

15. This package makes it easy to fit models (<https://github.com/paul-buerkner/brms>).


```

|-- modelcode.stan <- Model definition.
|-- notebooks/ <- Jupyter notebooks.
|-- rosi_py.ipynb
|-- rosi_py_files/ <- Subdirectory for temporary
                    outputs such as figures.
|-- README.md <- Documentation for this
                    subdirectory.

```

We have found this directory structure to be very helpful and useful in the case of an exploratory data analysis project. Additionally, there is value in reusing the same structure for other projects (given a structure that works for us): By reducing unnecessary cognitive load, this practice has made our day-to-day more productive and more enjoyable. For further inspiration, we refer the interested reader to [Tra17], [Dc] and references therein.

The second step we took was to set up the project as its own Git repository¹⁷. Thus, we can track changes conveniently and copy ('clone') the project on other machines safely (preserving the directory structure and, hence, relative paths)¹⁸.

Reproducible research practitioners recommend licensing your scientific work under a license which ensures attribution and facilitates sharing [Sto09]. Raw data are not copyrightable, so it makes no sense to license them. Code should be made available under a FLOSS license. Licenses suitable for materials which are neither software nor data (i.e., papers, reports, figures), and offering both attribution and ease of sharing, are the Creative Commons Attribution (CC BY) licenses. The case study (notebook) has been licensed under CC BY since the beginning. This practice can indeed contribute to improving reproducibility, since other researchers may then reuse the materials independently, without having to ask the copyright holders for permission.

We were confronted with the issue of software portability in real life, as soon as we (the authors) started collaborating. We created an isolated Python 3 environment with *conda*, a cross-platform package and environment manager¹⁹. As it turned out, the conventional file `environment.yml`, which specifies package dependencies, did not suffice: We run different operating systems and some dependencies were not available for the other platform. Therefore, we included a `spec-file.txt` as a specification file for creating the *conda* environment on GNU/Linux. Admittedly, this feels only mildly satisfying and we would welcome feedback from the community.

At the moment, all the analysis takes place in one long Jupyter notebook²⁰. We could break it down into smaller notebooks (and name them with number prefixes, for ordering). This way, someone new to the project could identify the various modelling and computing steps, in order, only by looking at the 'self-documenting' file structure. If we ever take the project to a production-like stage, we could further modularize the functionality of each notebook into modules (.py files), which would contain functions and would be organized into a project-specific Python package. This would pave the way for creating a build file²¹ which would chain all operations together and generate results for our specific project. Reaching this stage is referred to as *automation*.

17. Git is a distributed version control system which is extremely popular in software development (<https://git-scm.com/>).

18. The *mixed-data* project is hosted remotely at <https://github.com/bayesways/mixed-data>.

19. See <https://conda.io/docs/>.

20. See https://github.com/bayesways/mixed-data/blob/2f4e4ea72466a4884dc2a5c46510129fac602f1f/notebooks/rosi_py.ipynb.

21. See <https://swcarpentry.github.io/make-novice/reference#build-file>.

In data analysis, the first of these operations usually consists in accessing the initial, raw dataset(s). This brings about the question of data availability. In human subject research, such as clinical trials, the raw data cannot, and should not, be made publicly available. We acknowledge the tension existing between reproducibility and privacy²². At the time of this writing and as mentioned in the case study section, we are showcasing the analysis only with synthetic input data.

REFERENCES

- [Bar18] Pablo Barberá. *The Trade-Off Between Reproducibility and Privacy in the Use of Social Media Data to Study Political Behavior*. University of California Press, Oakland, CA, 2018. URL: <https://www.practicereproducibleresearch.org/case-studies/barbera.html>.
- [BG13] Michael Betancourt and Mark Girolami. Hamiltonian monte carlo for hierarchical models. 2013. [arXiv:1312.0906v1](https://arxiv.org/abs/1312.0906v1).
- [CG98] Siddhartha Chib and Edward Greenberg. Analysis of multivariate probit models. *Biometrika*, 85(2):347–361, jun 1998. doi:10.1093/biomet/85.2.347.
- [Dc] DrivenData and contributors. The cookiecutter data science project. Accessed on Wed, May 23, 2018. URL: <http://drivendata.github.io/cookiecutter-data-science/>.
- [GCS+13] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. CRC press, 2013.
- [Kit18] Justin Kitzes. *The Basic Reproducible Workflow Template*, chapter 3. University of California Press, Oakland, CA, 2018. URL: <https://www.practicereproducibleresearch.org/core-chapters/3-basic.html>.
- [KTD18] J. Kitzes, D. Turek, and F. Deniz, editors. *The Practice of Reproducible Research: Case Studies and Lessons from the Data-Intensive Sciences*. University of California Press, Oakland, CA, 2018. URL: <https://www.practicereproducibleresearch.org/>.
- [McE15] Richard McElreath. *Statistical rethinking : a Bayesian course with examples in R and Stan*. 2015.
- [McE17] Richard McElreath. Markov chains: Why walk when you can fly?. 2017. Accessed on Wed, May 23, 2018. URL: <http://eleventh.org/blog/2017/11/28/build-a-better-markov-chain/>.
- [MR06] Jean-Michel Marin and Christian P. Robert. *The bayesian core: a practical approach for computational bayesian statistics*, volume 102. Springer Texts in Statistics, 2006. doi:10.1016/j.peva.2007.06.006.
- [RMS18] Ariel Rokem, Ben Marwick, and Valentina Staneva. *Assessing Reproducibility*, chapter 2. University of California Press, Oakland, CA, 2018. URL: <https://www.practicereproducibleresearch.org/core-chapters/2-assessment.html>.
- [Rob07] Christian P. Robert. *The Bayesian choice: from decision-theoretic foundations to computational implementation*. Springer Science & Business Media, 2007.
- [Sto09] Victoria Stodden. Enabling reproducible research: Open licensing for scientific innovation. *International Journal of Communications Law and Policy*, 2009. doi:10.7916/D8N01H1Z.
- [TDM12] Aline Talhouk, Arnaud Doucet, and Kevin Murphy. Efficient bayesian inference for multivariate probit models with sparse inverse correlation matrices. *Journal of Computational and Graphical Statistics*, 21(3):739–757, jul 2012. doi:10.1080/10618600.2012.679239.
- [Tea17] Stan Development Team. Stan modeling language: User's guide and reference manual, 2017. URL: <https://github.com/stan-dev/stan/releases/download/v2.17.0/stan-reference-2.17.0.pdf>.
- [Tra17] Dustin Tran. A research to engineering workflow, 2017. Accessed on Wed, May 23, 2018. URL: <http://dustintran.com/blog/a-research-to-engineering-workflow>.
- [Woo15] Simon N. Wood. *Core Statistics*. Cambridge University Press, 2015. URL: <https://people.maths.bris.ac.uk/~sw15190/core-statistics.pdf>.

22. A case study in political science is discussed in this respect in [Bar18]. Some private communication with political scientists and various technologists have led us to throw the idea of leveraging the blockchain to improve reproducibility in human subject research: What if the raw datasets could live as private data on a public blockchain, notably removing the possibility of cherry-picking by design?

Scalable Feature Extraction with Aerial and Satellite Imagery

Virginia Ng^{‡*}, Daniel Hofmann[‡]

<https://youtu.be/3AuRW9kq89g>

Abstract—Deep learning techniques have greatly advanced the performance of the already rapidly developing field of computer vision, which powers a variety of emerging technologies—from facial recognition to augmented reality to self-driving cars. The remote sensing and mapping communities are particularly interested in extracting, understanding and mapping physical elements in the landscape. These mappable physical elements are called features, and can include both natural and synthetic objects of any scale, complexity and character. Points or polygons representing sidewalks, glaciers, playgrounds, entire cities, and bicycles are all examples of features. In this paper we present a method to develop deep learning tools and pipelines that generate features from aerial and satellite imagery at large scale. Practical applications include object detection, semantic segmentation and automatic mapping of general-interest features such as turn lane markings on roads, parking lots, roads, water, building footprints.

We give an overview of our data preparation process, in which data from the Mapbox Satellite layer, a global imagery collection, is annotated with labels created from OpenStreetMap data using minimal manual effort. We then discuss the implementation of various state-of-the-art detection and semantic segmentation systems such as the improved version of You Only Look Once (YOLOv2), modified U-Net, Pyramid Scene Parsing Network (PSPNet), as well as specific adaptations for the aerial and satellite imagery domain. We conclude by discussing our ongoing efforts in improving our models and expanding their applicability across classes of features, geographical regions, and relatively novel data sources such as street-level and drone imagery.

Index Terms—computer vision, deep learning, neural networks, satellite imagery, aerial imagery

I. Introduction

Location data is built into the fabric of our daily experiences, and is more important than ever with the introduction of new location-based technologies such as self-driving cars. Mapping communities, open source or proprietary, work to find, understand and map elements of the physical landscape. However, mappable physical elements are continually appearing, changing, and disappearing. For example, more than 1.2 million residential units were built in the United States alone in 2017 [buildings]. Therefore, a major challenge faced by mapping communities is maintaining recency while expanding worldwide coverage. To increase the speed and accuracy of mapping, allowing better pace-keeping with change

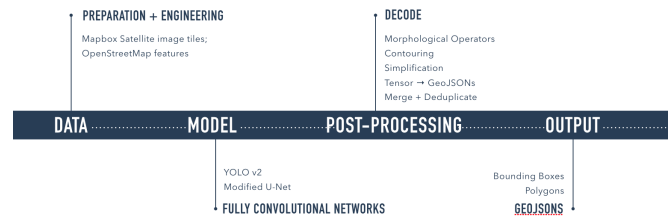


Fig. 1: Computer Vision Pipeline.

in the mappable landscape, we propose integrating deep neural network models into the mapping workflow. In particular, we have developed tools and pipelines to detect various geospatial features from satellite and aerial imagery at scale. We collaborate with the OpenStreetMap [osm] (OSM) community to create reliable geospatial datasets, validated by trained and local mappers.

Here we present two use cases to demonstrate our workflow for extracting street navigation indicators such as turn restrictions signs, turn lane markings, and parking lots, in order to improve our routing engines. Our processing pipelines and tools are designed with open source libraries including Scipy, Rasterio, Fiona, Osum, JOSM, Keras, PyTorch, and OpenCV, while our training data is compiled from OpenStreetMap and the Mapbox Maps API [mapbox_api]. Our tools are designed to be generalizable across geospatial feature classes and across data sources.

II. Scalable Computer Vision Pipelines

The general design for our deep learning based computer vision pipelines can be found in Figure 1, and is applicable to both object detection and semantic segmentation tasks. We design such pipelines with two things in mind: they must scale to process petabytes worth of data; and they must be agile enough to be repurposed for computer vision tasks on other geospatial features. This requires tools and libraries that make up these pipelines to be developed in modularized fashion. We present turn lane markings as an example of an object detection pipeline, and parking lots as an example of a semantic segmentation pipeline. Code for Robosat [robosat], our end-to-end semantic segmentation pipeline, along with all its tools, is made available at: <https://github.com/mapbox/robosat>.

* Corresponding author: virginia@mapbox.com

‡ Mapbox

Copyright © 2018 Virginia Ng et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

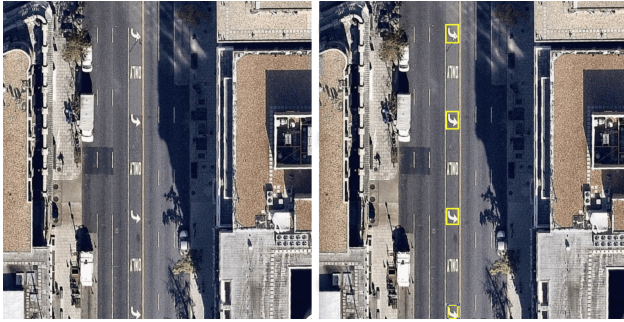


Fig. 2: Left: Original satellite image. Right: Turn lane markings detection.

1. Data

The data needed to create training sets depends on the type of task: object detection or semantic segmentation. We first present our data preparation process for object detection and then discuss the data preparation process for semantic segmentation.

Data Preparation For Object Detection. Object detection is the computer vision task that deals with locating and classifying a variable number of objects in an image. Figure 2 demonstrates how object detection models are used to classify and locate turn lane markings from satellite imagery. There are many other practical applications of object detection such as face detection, counting, and visual search engines. In our case, detected turn lane markings become valuable navigation assets to our routing engines when determining the most optimal routes.

The turn lane marking training set is created by collecting imagery of various types of turn lane markings and manually drawing a bounding box around each marking. We use Overpass Turbo¹ to query the OpenStreetMap database for streets containing turn lane markings, i.e., those tagged with one of the following attributes: “turn:lane=*”, “turn:lane:forward=*”, “turn:lane:backward=*” in OpenStreetMap. The marked street segments, as shown in Figure 3, are stored as GeoJSON features clipped into the tiling scheme [tile] of the Mapbox Satellite basemap [mapbox]. Figure 4 shows how skilled mappers use this map layer as a cue to manually draw bounding boxes around each turn lane marking using JOSM², a process called annotation. These bounding boxes are stored in GeoJSON polygon format on Amazon S3 [s3] and used as labels during training.

Mappers annotate over 54,000 turn lane markings, spanning six classes - “Left”, “Right”, “Through”, “ThroughLeft”, “ThroughRight”, and “Other” in five cities. Turn lane markings of all shapes and sizes, as well as ones that are partially covered by cars and/or shadows are included in this training set. To ensure a high-quality training set, we had a separate group of mappers verify each of the bounding boxes drawn. We exclude turn lane markings that are not visible, as seen in Figure 5.

Data Engineering Pipeline for Object Detection. Within the larger object detection pipeline, sits a data engineering pipeline

1. JOSM [josm] is an extensible OpenStreetMap editor for Java 8+. At its core, it is an interface for editing OSM, i.e., manipulating the nodes, ways, relations, and tags that compose the OSM database. Compared to other OSM editors, JOSM is notable for its range of features, such as allowing the user to load arbitrary GPX tracks, background imagery, and OpenStreetMap data from local and online sources. It is open source and licensed under GPL.

2. Overpass Turbo [overpass] is a web based data mining tool for OpenStreetMap. It runs any kind of Overpass API query and shows the results on an interactive map.



Fig. 3: A custom layer created by clipping the locations of roads with turn lane markings to Mapbox Satellite. Streets with turn lane markings are rendered in red.



Fig. 4: Annotating turn lane markings by drawing bounding boxes.

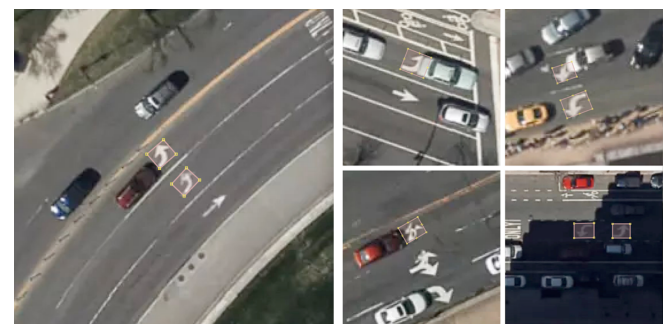


Fig. 5: Left: Examples of visible turn lane markings that are included in the training set. Right: Defaced or obscured turn lane markings, such as those covered by cars, are excluded from the training set.

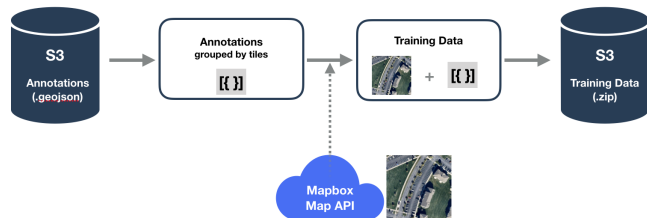


Fig. 6: Object Detection Data Engineering Pipeline: Annotated OpenStreetMap GeoJSON features are converted to image pixel space, stored as JSON image attributes and used as training labels. These labels are then combined with each of their respective imagery tiles, fetched from the Mapbox Maps API (Satellite), to create a training set for turn lane marking detection.



Fig. 7: Left: Original satellite image. Right: Semantic segmentation of roads, buildings and vegetation.

designed to create and process training data in large quantities. This data engineering pipeline is capable of streaming any set of prefixes off of Amazon S3 into prepared training sets. Several pre-processing steps are taken to convert annotations to the appropriate data storage format before combining them with real imagery. The turn lane marking annotations are initially stored as GeoJSON polygons grouped by class. Each of these polygons is streamed out of the GeoJSON files on S3, converted to image pixel coordinates, and stored as JSON image attributes to abstract tiles [tile]. The pre-processed annotations are randomly assigned to training and testing datasets with a ratio of 4:1. The abstract tiles are then replaced by the corresponding real image tiles, fetched from the Satellite layer of the Mapbox Maps API. At this point, each training sample consisted of a photographic image paired with its corresponding JSON image attribute. Finally, the training and test sets are zipped and uploaded to Amazon S3. This process is scaled up to run multiple cities in parallel on Amazon Elastic Container Service³. This data engineering pipeline is shown in Figure 6.

Data Preparation for Semantic Segmentation. Semantic segmentation is the computer vision task that partitions an image into semantically meaningful parts, and classifies each part into one of any pre-determined classes. This can be understood as assigning a class to each pixel in the image, or equivalently as drawing non-overlapping masks or polygons with associated classes over the image. As an example of the polygonal approach, in addition to distinguishing roads from buildings and vegetation, we also delineate the boundaries of each object in Figure 7.

The parking lot training set is created by combining imagery tiles collected from Mapbox Satellite with parking lots polygons. Parking lot polygons are generated by querying the OpenStreetMap database with Osmium [osmium] for OpenStreetMap features with attributes “tag:amenity=parking=*” using the *rs-ex-*

tract tool [rs-extract] in Robosat, our segmentation pipeline. These parking lot polygons are stored as two-dimensional single-channel numpy arrays, or binary mask clipped and scaled to the Mapbox Satellite tiling scheme using the *rs-rasterize* tool [rs-rasterize]. Each mask array is paired with its corresponding photographic image tile. Conceptually, this can be compared to concatenating a fourth channel, the mask, onto a standard red, green, and blue image. 55,710 parking lots are annotated for the initial training set. Our tools and processes can be generalized to any OpenStreetMap feature and any data source. For example, we also experiment with building segmentation in unmanned aerial vehicle (UAV) imagery from the OpenAerialMap project in Tanzania [tanzania]. One can generate training sets for any OpenStreetMap feature in this way by writing custom Osmium handlers to convert OpenStreetMap geometries into polygons.

2. Model

Fully Convolutional Neural Networks. Fully convolutional networks (FCNs) are neural networks composed only of convolutional layers. They are contrasted with more conventional networks that typically have fully connected layers or other non-convolutional subarchitectures as “decision-makers” just before the output. For the purposes considered here, FCNs show several significant advantages. First, FCNs can handle input images of different resolutions, while most alternatives require input dimensions to be of a certain size [FCN]. For example, architectures like AlexNet can only work with input images sizes that are 224 x 224 x 3 [FCN]. Second, FCNs are well suited to handling spatially dense prediction tasks like segmentation because one would no longer be constrained by the number of object categories or complexity of the scenes. Networks with fully connect layers, in contrast, generally lose spatial information in these layers because all output neurons are connected to all input neurons [FCN].

Object Detection Models. Many of our applications require low latency prediction from their object detection algorithms. We implement YOLOv2 [yolov2], the improved version of the real-time object detection system You Only Look Once (YOLO) [yolo], in our turn lane markings detection pipeline. YOLOv2 outperforms other state-of-the-art methods, like Faster R-CNN with ResNet [resnet] and Single Shot MultiBox Detector (SSD) [ssd], in both speed and detection accuracy [yolov2]. It works by first dividing the input image into 13 x 13 grid cells (i.e., there are 169 total cells for any input image). Each grid cell is responsible for generating 5 bounding boxes. Each bounding box is composed of its center coordinates relative to the location of its corresponding grid cell, its normalized width and height, a confidence score for “objectness,” and an array of class probabilities. A logistic activation is used to constrain the network’s location prediction to fall between 0 and 1, so that the network is more stable. The objectness predicts the intersection over union (IOU) of the ground truth and the proposed box. The class probabilities predict the conditional probability of each class for the proposed object, given that there is an object in the box [yolov2].

6 classes are defined for the turn lane markings detection project. With 4 coordinates defining each box’s geometry, the

³. Osmium [osmium] is a fast and flexible C++ library for working with OpenStreetMap data.

⁴. Amazon ECS [ecs] is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on specified type of instances

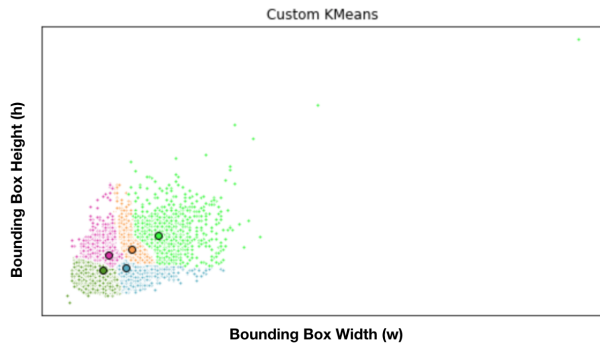


Fig. 8: Clustering of box dimensions in the turn lane marking training set. We run *k*-means clustering on the dimensions of bounding boxes to get anchor boxes for our model. We used $k = 5$, as suggested by the YOLOv2 authors, who found that this cluster count gives a good tradeoff for recall v. complexity of the model.

"objectness" confidence, and 6 class probabilities, each bounding box object is comprised of 11 numbers. Multiplying by boxes per grid cell and grid cells per image, this project's YOLOv2 network therefore always yields $13 \times 13 \times 5 \times 11 = 9,295$ outputs per image.

The base feature extractor of YOLOv2 is Darknet-19 [darknet], a FCN composed of 19 convolutional layers and 5 maxpooling layers. Detection is done by replacing the last convolutional layer of Darknet-19 with three 3×3 convolutional layers, each outputting 1024 channels. A final 1×1 convolutional layer is then applied to convert the $13 \times 13 \times 1024$ output into $13 \times 13 \times 55$. We follow two suggestions proposed by the YOLOv2 authors when designing our model. The first is incorporating batch normalization after every convolutional layer. During batch normalization, the output of a previous activation layer is normalized by subtracting the batch mean and dividing by the batch standard deviation. This technique stabilizes training, improves the model convergence, and regularizes the model [yolov2_batch]. By including batch normalization, YOLOv2 authors saw a 2% improvement in mAP on the VOC2007 dataset [yolov2] compared to the original YOLO model. The second suggestion is the use of anchor boxes and dimension clusters to predict the actual bounding box of the object. This step is achieved by running *k*-means clustering on the turn lane marking training set bounding boxes. As seen in Figure 8, the ground truth bounding boxes for turn lane markings follow specific height-width ratios. Instead of directly predicting bounding box coordinates, our model predicts the width and height of the box as offsets from cluster centroids. The center coordinates of the box relative to the location of filter application is predicted by using a sigmoid function.

Our model is first pre-trained on ImageNet 224×224 resolution imagery. The network is then resized and fine-tuned for classification on 448×448 turn lane marking imagery, to ensure that the relatively small features of interest are still reliably detected.

Segmentation Models. For parking lot segmentation, we select an approach of binary segmentation (distinguishing parking lots from the background), and found U-Net [unet] to be a suitable architecture. The U-Net architecture can be found in Figure 9. It consists of a contracting path, to capture context, and a symmetric expanding path, which allows precise localization. This type of network can be trained end-to-end with very few training images

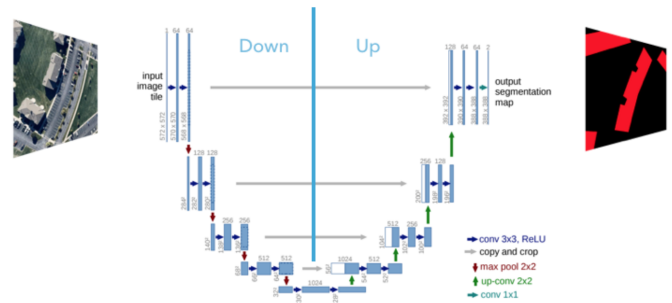


Fig. 9: U-Net architecture.

and can yield more precise segmentations than prior state-of-the-art methods such as sliding-window convolutional networks. The first part of the U-Net network downsamples, and is similar in design and purpose to the encoding part of an autoencoder. It repeatedly applies convolution blocks followed by maxpool downsamplings, encoding the input image into increasingly abstract representations at successively deeper levels. The second part of the network consists of upsampling and concatenation, followed by ordinary convolution operations. Concatenation combines relatively "raw" information with relatively "processed" information. This can be understood as allowing the network to assign a class to a pixel with sensitivity to small-scale, less-abstract information about the pixel and its immediate neighborhood (e.g., whether it is gray) and simultaneously with sensitivity to large-scale, more-abstract information about the pixel's context (e.g., whether there are nearby cars aligned in the patterns typical of parking lots). we gain a modest 1% improvement in accuracy by making two additional changes. First we replace the standard U-Net encoder with pre-trained ResNet50 [resnet] encoder. Then, we switch out the learned deconvolutions with nearest neighbor upsampling followed by a convolution for refinement.

We experiment with a Pyramid Scene Parsing Network (PSPNet) [pspnet] architecture for a 4-class segmentation task on buildings, roads, water, and vegetation. PSPNet is one of the few pixel-wise segmentation methods that focuses on global priors, while most methods fuse low-level, high resolution features with high-level, low resolution ones to develop comprehensive feature representations. Global priors can be especially useful for objects that have similar spatial features. For instance, runways and freeways have similar color and texture features, but they belong to different classes, which can be discriminated by adding car and building information. PSPNet uses pre-trained ResNet to generate a feature map that is $1/8$ the size of the input image. The feature map is then fed through the pyramid parsing module, a hierarchical global prior that aggregates different scales of information. After upsampling and concatenation, the final feature representation is fused with a 3×3 convolution to produce the final prediction map. As seen in Figure 6, PSPNet produced good-quality segmentation masks in our tests on scenes with complex features such as irregularly shaped trees, buildings and roads. For the 2-class parking lot task, however, we found PSPNet unnecessarily complex and time-consuming.

Hard Negative Mining. This is a technique we apply to improve model accuracy [hnm]. We first train a model with an

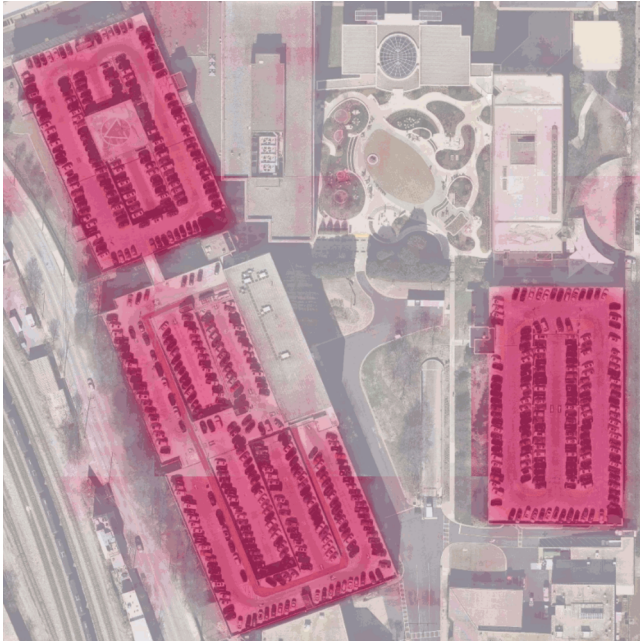


Fig. 10: A probability mask marking the pixels that our model believes belong to parking lots.

initial subset of negative examples, and collect negative examples that are incorrectly classified by this initial model to form a set of hard negatives. A new model is then trained with the hard negative examples and the process may be repeated a few times.

Figure 10 shows a model's output as a probability mask overlaid on Mapbox Satellite. Increasingly opaque red indicates an increasingly high probability estimate of the underlying pixel belonging to a parking lot. We use this type of visualization to find representative falsely detected patches for use as hard negatives in hard negative mining.

3. Post-Processing

Figure 11 shows an example of the raw segmentation mask derived from our U-Net model. It cannot be used directly as input for OpenStreetMap. We perform a series of post-processing steps to refine and transform the mask until it met quality and format requirements for OpenStreetMap consumption:

Noise Removal. Noise in the output mask is removed by two morphological operations: erosion followed by dilation. Erosion removes some positive speckle noise ("islands"), but it also shrinks objects. Dilation re-expands the objects.

Fill in holes. The converse of the previous step, removing "lakes" (small false or topologically inconvenient negatives) in the mask.

Contouring. During this step, continuous pixels having same color or intensity along the boundary of the mask are joined. The output is a binary mask with contours.

Simplification. We apply Douglas-Peucker simplification [DP], which takes a curve composed of line segments and gives a similar curve with fewer vertices. OpenStreetMap favors polygons with the least number of vertices necessary to represent the ground truth accurately, so this step is important to increase the data's quality as perceived by its end users.

Transform Data. Polygons are converted from in-tile pixel coordinates to GeoJSONs in geographic coordinates (longitude and latitude).



Fig. 11: An example of border artifacts and holes in raw segmentation masks produced by our U-Net model.



Fig. 12: Left: Polygons crossing tile boundaries, and other adjacent polygons, are combined. Right: Combined polygons.

Merging multiple polygons. This tool combines polygons that are nearly overlapping, such as those that represent a single feature broken by tile boundaries, into a single polygon. See Figure 12.

Deduplication. Cleaned GeoJSON polygons are compared against parking lot polygons that already exist in OpenStreetMap, so that only previously unmapped features are uploaded.

All post-processing tools can be found in our Robosat [robosat] GitHub repository.

4. Conclusion

We demonstrated the steps to building deep learning-based computer vision pipelines that can run object detection and segmentation tasks at scale. With these pipeline designs, we are able to create training data with minimal manual effort, experiment with different network architectures, run inference, and apply post-process algorithms to tens of thousands of image tiles in parallel using Amazon ECS. The outputs of the processing pipelines discussed are turn lane markings and parking lots in the form of GeoJSON features suitable for adding to OpenStreetMap. Mapbox routing engines then take these OpenStreetMap features into

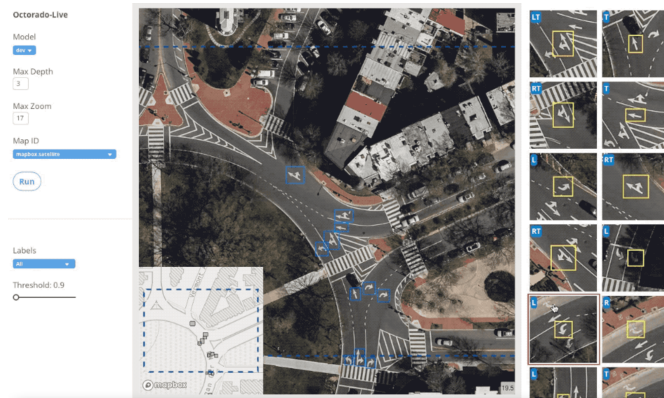


Fig. 13: Front-end UI for instant turn lane marking detection on Mapbox Satellite layer, a global imagery collection.

account when calculating optimal navigation routes. As we make various improvements to our baseline model and post-processing algorithms (see below), we keep human control over the final decision to add a given feature to OpenStreetMap. Figure 13 shows a front-end user interface (UI) created to allow users to run instant turn lane marking detection and visualize the results on top of Mapbox Satellite. Users can select a model, adjust the level of confidence for the model, choose from any Mapbox map styles [mapbox_style], and determine the area on the map to run inference on [mapbox_zoom].

IV. Future Work

We are now working on making a few improvements to Robosat, our segmentation pipeline, so that it becomes more flexible in handling input image of different resolutions. First, our existing post-processing handler is designed for parking lot features and is specifically tuned with thresholds set for zoom level 18 imagery [osm_zoom]. We are replacing these hard-coded thresholds with generalized ones that are calculated based on resolution in meters per pixel. We also plan to experiment with a feature pyramid-based deep convolutional network called Feature Pyramid Network (FPN) [FPN]. It is a practical and accurate solution to multi-scale object detection. Similar to U-Net, the FPN has lateral connections between the bottom-up pyramid (left) and the top-down pyramid (right). The main difference is where U-net only copies features and appends them, FPN applies a 1x1 convolution layer before adding the features. We will most likely follow the authors' footsteps and use ResNet as the backbone of this network.

There two other modifications planned for the post-processing steps. First, we want to experiment with a more sophisticated polygon simplification algorithm besides Douglas-Peucker. Second, we are rethinking the ordering of first performing simplification then merging. The current post-process workflow performs simplification on individual extracted polygons and then merges polygons that are across imagery tiles together. The resulting polygons, according to this process, may no longer be in the simplest shape.

We design our tools and pipelines with the intent that other practitioners would find it straightforward to adapt them to other landscapes, landscape features, and imagery data sources. For instance, we generated 184,000 turn restriction detections following a similar process applying deep learning models on Microsoft's street-level imagery [streetside]. We released these turn restriction detections located across 35,200 intersections and 23 cities for

the OpenStreetMap community [turn-restrict] in June 2018. For future work we will continue to look for ways to bring different sources and structures of data together to build better computer vision pipelines.

REFERENCES

- [buildings] Cornish, C., Cooper, S., Jenkins, S., & US Census Bureau. (2011, August 23). US Census Bureau New Residential Construction. Retrieved from <https://www.census.gov/construction/nrc/index.html>
- [osm] OpenStreetMap Contributors. (2017). OpenStreetMap. Retrieved May 30, 2018, from <https://www.openstreetmap.org/>
- [mapbox] Mapbox. (n.d.). About. Retrieved June 30, 2018, from <https://www.mapbox.com/about/>
- [mapbox_api] Mapbox. (n.d.). Mapbox API Documentation. Retrieved May 30, 2018, from <https://www.mapbox.com/api-documentation/#maps>
- [osm-lanes] OpenStreetMap Contributors. (2018, February 27). Lanes. Retrieved May 30, 2018, from <https://wiki.openstreetmap.org/wiki/Lanes>
- [overpass] Raifer, M. (2017, January). Overpass Turbo. Retrieved from <https://overpass-turbo.eu/>
- [josm] Scholz, I., & Stöcker, D. (2017, May). Java OpenStreetMap Editor. Retrieved from <https://josm.openstreetmap.de/>
- [osm-parking] OpenStreetMap Contributors. (2018, April). Tag:amenity=parking. Retrieved from <https://wiki.openstreetmap.org/wiki/Tag:amenity%3Dparking>
- [rs-extract] Mapbox. (2018, June). Robosat. Retrieved from <https://github.com/mapbox/robosat#rs-extract>
- [rs-rasterize] Mapbox. (2018, June). Robosat. Retrieved from <https://github.com/mapbox/robosat#rs-rasterize>
- [osmium] Topf, J. (2018, April). Osmcode/libosmium. Retrieved May 11, 2018, from <https://github.com/osmcode/libosmium>
- [tile] OpenStreetMap Contributors. (2018, June). Tile Scheme. Retrieved from https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames
- [tanzania] Hofmann, D. (2018, July 5). Daniel-j-h's diary | RoboSat loves Tanzania. Retrieved from <https://www.openstreetmap.org/user/daniel-j-h/diary/44321>
- [s3] Amazon. (n.d.). Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service. Retrieved from <https://aws.amazon.com/s3/>
- [ecs] Amazon. (n.d.). Amazon ECS - run containerized applications in production. Retrieved from <https://aws.amazon.com/ecs/>
- [yolo] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016, June). You Only Look Once: Unified, Real-Time Object Detection. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2016.91
- [ssd] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C., & Berg, A. C. (2016, September 17). SSD: Single Shot MultiBox Detector. Computer Vision – ECCV 2016 Lecture Notes in Computer Science, 21-37. doi:10.1007/978-3-319-46448-0_2
- [darknet] Redmon, J. (2013-2016). Darknet: Open Source Neural Networks in C. Retrieved from <https://pjreddie.com/darknet/>
- [yolov2] Redmon, J., & Farhadi, A. (2017, July). YOLO9000: Better, Faster, Stronger. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2017.690
- [yolov2_batch] Ioffe, S., & Szegedy, C. (2015, February 11). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167
- [FCN] Long, J., Shelhamer, E., & Darrell, T. (2015, June). Fully Convolutional Networks for Semantic Segmentation. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/CVPR.2015.7298965
- [unet] Ronneberger, O., Fischer, P., & Brox, T. (2015, May 18) U-Net: Convolutional Networks for Biomedical Image Segmentation. 2015 MICCAI. arXiv:1505.04597
- [resnet] He, K., Zhang, X., Ren, S., & Sun, J. (2016, June). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2016.90
- [pspnet] Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J. (2017, July). Pyramid Scene Parsing Network. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2017.660

- [hnm] Dalal, N., & Triggs, B. (2005, June). Histograms of oriented gradients for human detection. 2005 IEEE Conference on Computer Vision and Pattern Recognition. 10.1109/CVPR.2005.177
- [robosat] Mapbox. (2018, June). Robosat. Retrieved from <https://github.com/mapbox/robosat>
- [DP] Wu, S., & Marquez, M. (2003, October). A non-self-intersection Douglas-Peucker algorithm. 16th Brazilian Symposium on Computer Graphics and Image Processing (SIB-GRAPI 2003). doi:10.1109/sibgra.2003.1240992
- [mapbox_style] Mapbox. (n.d.). Styles. Retrieved from <https://www.mapbox.com/help/studio-manual-styles/>
- [mapbox_zoom] Mapbox. (n.d.). Zoom Level. Retrieved from <https://www.mapbox.com/help/define-zoom-level/>
- [osm_zoom] OpenStreetMap Contributors. (2018, June 20). Zoom Levels. Retrieved June 30, 2018, from https://wiki.openstreetmap.org/wiki/Zoom_levels
- [FPN] Lin, T., Dollar, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017, July). Feature Pyramid Networks for Object Detection. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2017.106
- [streetside] Microsoft. (n.d.). Streetside. Retrieved from <https://www.microsoft.com/en-us/maps/streetside>
- [turn-restrict] Ng, V. (2018, June 14). virginiajung's diary | Releasing 184K Turn Restriction Detections. Retrieved from <https://www.openstreetmap.org/user/virginiajung/diary/44171>

signac: A Python framework for data and workflow management

Vyas Ramasubramani^{‡*}, Carl S. Adorf[‡], Paul M. Dodd[‡], Bradley D. Dice[¶], Sharon C. Glotzer^{‡§¶||}

<https://youtu.be/CCKQH1M2uR4>

Abstract—Computational research requires versatile data and workflow management tools that can easily adapt to the highly dynamic requirements of scientific investigations. Many existing tools require strict adherence to a particular usage pattern, so researchers often use less robust ad hoc solutions that they find easier to adopt. The resulting data fragmentation and methodological incompatibilities significantly impede research. Our talk showcases *signac*, an open-source Python framework that offers highly modular and scalable solutions for this problem. Named for the Pointillist painter Paul Signac, the framework's powerful workflow management tools enable users to construct and automate workflows that transition seamlessly from laptops to HPC clusters. Crucially, the underlying data model is completely independent of the workflow. The flexible, serverless, and schema-free *signac* database can be introduced into other workflows with essentially no overhead and no recourse to the *signac* workflow model. Additionally, the data model's simplicity makes it easy to parse the underlying data without using *signac* at all. This modularity and simplicity eliminates significant barriers for consistent data management across projects, facilitating improved provenance management and data sharing with minimal overhead.

Index Terms—data management, database, data sharing, provenance, computational workflow, hpc

Introduction

Streamlining data generation and analysis is a critical challenge for science in the age of big data and high performance computing (HPC). Modern computational resources can generate and consume enormous quantities of data, but process automation and data management tools have lagged behind. The highly file-based workflows characteristic of computational science are not amenable to traditional relational databases, and HPC applications require that data is available on-demand, enforcing strict performance requirements for any data storage mechanism. Building processes acting on this data requires transparent interaction with HPC clusters without sacrificing testability on personal computers, and these processes must be sufficiently malleable to adapt to changes in scientific inquiries.

* Corresponding author: vramasub@umich.edu

‡ Department of Chemical Engineering, University of Michigan, Ann Arbor

¶ Department of Physics, University of Michigan, Ann Arbor

§ Department of Materials Science and Engineering, University of Michigan, Ann Arbor

|| Biointerfacing Institute, University of Michigan, Ann Arbor

Copyright © 2018 Vyas Ramasubramani et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

To illustrate the obstacles that must be overcome, we consider a simple example in which we study the motion of an object through a fluid medium. If we initially model the motion only as a function of one parameter, an ad hoc solution for data storage would be to store the trajectories in paths named for the values of this parameter. If we then introduce some post-processing step, we could run it on each of these files. However, a problem arises if we realize that some additional parameter is also relevant. A simple solution might be to just rename the files to account for this parameter as well, but this approach would quickly become intractable if the parameter space increased further. A more flexible traditional solution involving the use of a relational MySQL [Cor16] database, for instance, might introduce undesirable setup costs and performance bottlenecks for file-based workflows on HPC. Even if we do employ such a solution, we also have to account for our workflow process: we need a way to run analysis and post-processing on just the new data points without performing unnecessary work on the old ones.

This paper showcases the *signac* framework, a data and workflow management tool that addresses these issues in a simple, powerful, and flexible manner (Fig. 1). The framework derives its name from the painter Paul Signac, one of the early pioneers of the Pointillist painting style. This style, in which paintings are composed of individual points of color rather than brushstrokes, provides an apt analogy for the underlying data model of the *signac* framework in which a data space is composed of individual data points that must be viewed together to make a complete picture. By storing JSON-encoded [Ecm17] metadata and the associated data together directly on the file system, *signac* provides database functionality such as searching and grouping data without the overhead of maintaining a server or interfacing with external systems, and it takes advantage of the high performance file systems common to HPC. Additionally, a *signac* database is entirely contained within a single root directory, making it compact and highly portable.

With *signac*, data space modifications like the one discussed above are trivially achievable with just a few lines of Python code. *signac*'s workflow component makes it just as easy to modify the process of data generation by simply defining the steps as Python functions. The workflow component of the framework, *signac-flow*, will immediately enable the use of these functions on the existing data space through a single command, and it tracks which tasks are completed to avoid redundancy. The resulting data can be accessed without reference to the workflow, ensuring that it is immediately available to anyone irrespective of

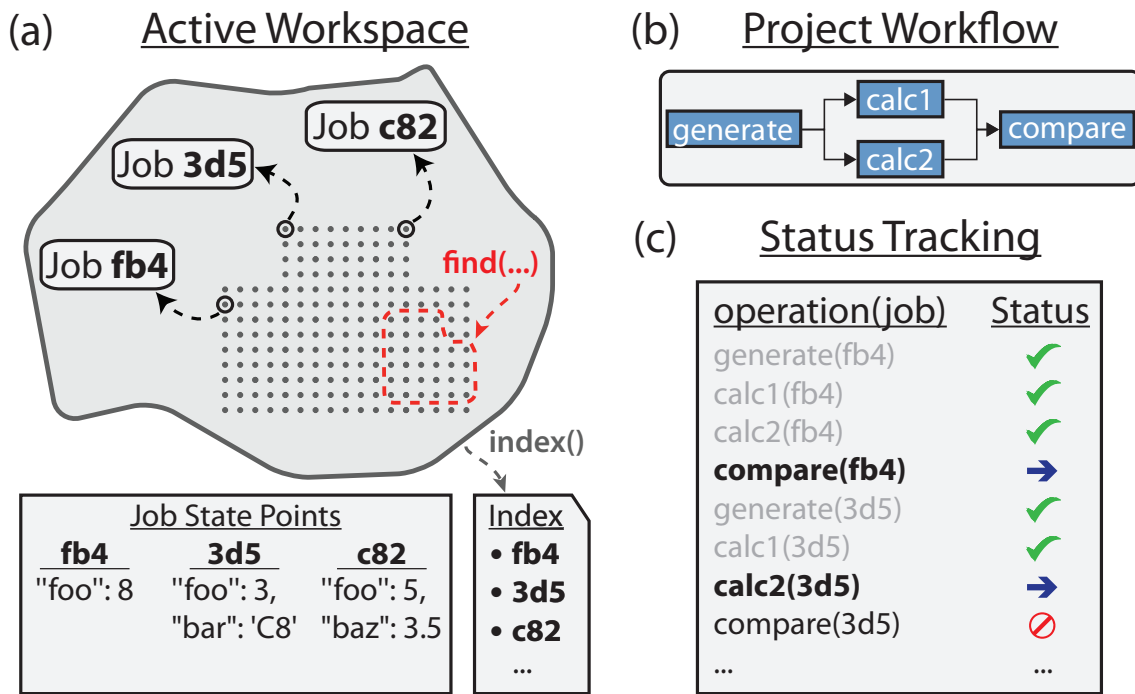


Fig. 1: The data in a *signac* project (a) is contained in its workspace (dark grey outline), which in turn is composed of individual data points (grey points) that exist within some multidimensional parameter space (light grey background). Each data point, or job, is associated with a unique hash value (e.g., 3d5) computed from its state point, the unique key identifying the job. Using *signac*, the data can be easily searched, filtered, grouped, and indexed. To generate and act on this data space, *signac* can be used to define workflows (b), which are generically represented as a set of operations composing a directed graph. Using a series of pre- and post-conditions defined on these operations, *signac* tracks the progress of this workflow on a per-job basis (c) to determine whether a particular job is complete (greyed text, green check), eligible (bold text, blue arrow), or blocked (normal text, universal no).

the tools they are using.

Overview and Examples

To demonstrate how *signac* works, we take a simple, concrete example of the scenario described above. Consider an experiment in which we want to find the optimal launch angle to maximize the distance traveled by a projectile through air. Figure 2 shows how we might organize the data associated with this study using *signac*. The central object in the *signac* data model is the *project*, which represents all the data associated with a particular instance of a *signac* data space. All of the project's data is contained within the *workspace* directory (see also Fig. 1). The workspace holds subdirectories corresponding to *jobs*, which are the individual data points in the data space. Each job is uniquely identified by its *state point*, which is an arbitrary key-value mapping. Although we see that these objects are stored in files and folders, we will show that these objects are structured in a way that provides layers of abstraction, making them far more useful than simple file system storage.

One could easily imagine interfacing existing scripts with this data model. The only requirement is some concept of a unique key for all data so that it can be inserted into the database. The unique key is what enables the creation of the 32 character hash, or *job id*, used to identify the job and its workspace folder (shown in Fig. 2). The uniqueness of this hash value is what enables *signac*'s efficient indexing and searching functionality. Additionally, this hash value is automatically updated to reflect any changes to

individual jobs, making them highly mutable. For example, if we instead wanted to consider how changing initial velocity affects the distance traveled for a particular angle, we can add the velocity to the existing job state points by taking advantage of the fact that the project object is an iterable:

```
for job in project:
    job.sp.v = 1
```

In this case, we wanted to modify the entire workspace; more generally, however, we might want to modify only some subset of jobs. One way to accomplish this would be to apply a filter within the loop using conditionals based on the job state point, e.g. `if job.sp.theta < 5: job.sp.v = 1`. A more elegant solution, however, is to take advantage of *signac*'s query API, which allows the user to find only the jobs of interest using a dictionary as a filter. For example, in the above snippet we could replace `for job in project` with `for job in project.find_jobs()`, using an arbitrary dictionary as the argument to `find_jobs()`, using an arbitrary dictionary as the argument to `find_jobs` to filter on the state point keys. The job finding functionality of *signac* is the entry point for its database functionality, which includes advanced indexing, selection, and grouping operations.

Having made the above change to our data space, we could now easily add new data points to test:

```
from numpy import linspace
for v in [1, 2, 3]:
    for theta in np.round(linspace(0, 1.57, 5), 2):
        sp = {"v": v, "theta": theta}
        project.open_job(sp).init()
```

```
In [1]: import signac
project = signac.init_project("Projectiles")
!ls
```

```
Notebook.ipynb  signac.rc
```

```
In [2]: job = project.open_job({"theta": 1.57})
job.init()
!find . -not -path '*/*.*'
```

```
.
./Notebook.ipynb
./signac.rc
./workspace
./workspace/4f8a64741a09749ac1320f4b61292e0c
./workspace/4f8a64741a09749ac1320f4b61292e0c/signac_statepoint.json
```

```
In [3]: print(job.get_id())
print(job.statepoint())
```

```
4f8a64741a09749ac1320f4b61292e0c
{'theta': 1.57}
```

Fig. 2: A very simple example using *signac* to create the basics of a data space. Initializing the project creates a *signac.rc* file, a configuration file identifying this folder as a *signac* project. The workspace directory is created when the first job is added to the project, and all job data is then stored in a subdirectory of the workspace. This subdirectory is named according to the job id, which is computed as the hash of the job state point. In this example, all work is conducted inside a Jupyter [PG07], [KRKP⁺16] notebook to indicate how easily this can be done. Note how fewer than ten lines of code are required to initialize a database and add data.

Jobs that already exist in the data space will not be overwritten by the *init* operation, so there is no harm in performing a loop like this multiple times.

All of *signac*'s core functionality is not only available as a Python library, but also as a command line tool. This tool uses the Python `setuptools console_scripts` entry point, so it is automatically installed with *signac* and ships with built-in help information. This interface not only facilitates the integration of *signac* with non-Python code bases and workflows, it is also very useful for more ad hoc analyses of *signac* data spaces. For example, searching the database using the command line can be very useful for quick data inspection:

```
$ # Many simple queries are automatically
$ # translated into JSON
$ signac find theta 0.39
Interpreted filter arguments as '{"theta": 0.39}'.
d3012d490304c3c1171a273a50b653ad
1524633c646adce7579abdd9c0154d0f
22fa30ddf3cc90b1b79d19fa7385bc95

$ # Operators (e.g. less than) are available
$ # using a "."-operator" syntax
$ signac find v.< 2
d61ac71a00bf73a38434c884c0aa82c9
00e5f0c36294f0eee4a30cab7c6046c
585599fe9149eed3e2dced76ef246903
22fa30ddf3cc90b1b79d19fa7385bc95
9fa1900a378aa05b9fd3d89f11ef0e5b
```

```
$ # More complex queries can be constructed
$ # using JSON directly
$ signac find '{"theta": {"$in": [0, 0.78]}}'
2faf0f76bde3af984a91b5e42e0d6a0b
585599fe9149eed3e2dced76ef246903
03d50a048c0423bda80c9a56e939f05b
3201fd381819dde4329d1754233f7b76
```

```
d61ac71a00bf73a38434c884c0aa82c9
13d54ee5821a739d50fc824214ae9a60
```

The query syntax is based on the MongoDB [Mon16] syntax and enables, for instance, logical and arithmetic operators. In fact, *signac* natively supports export of its databases to MongoDB. Although we can add support for integration with any database management system, we started with MongoDB for two reasons: first, because researchers are likely to prefer the comparatively less rigid approach of NoSQL databases to table-based relational databases; and second, because translation from a *signac* database to another JSON-based database is relatively straightforward. Due to the ease of export and shared query syntax, switching between *signac* and MongoDB is quite easy.

At any point, we can also get an overview of what the implicit data space schema looks like:

```
$ signac schema
{
  'theta': 'float([0.0, ..., 1.57], 5)',
  'v': 'int([1, 2, 3], 3)',
}
```

Keys with constant values across the entire data space can be optionally omitted from the schema. Additionally, schema can be filtered, nested keys can be compressed to specified depths, and the number of entries shown in each range can be limited as desired.

Workflows

The *signac* database is intended to be usable as a drop-in solution for data management issues. The *signac* framework, however, is designed to simplify the entire process of data generation, which includes clearly defining the processes that generate

and operate on the data cleanly and concisely. To manage workflows, the `signac-flow` component of the framework provides the `FlowProject` class (not to be confused with the `signac Project` class that interfaces with the data in a `signac` project). The `FlowProject` encodes operations acting on `signac` data spaces as well as the sequence information required to string these operations together into a complete workflow. In Fig. 3, we demonstrate how `signac-flow` can be used to automate our projectile investigation.

In this script, we register a simple function `calculate` as an operation with the `FlowProject.operation` decorator. We store our output in the *job document*, a lightweight JSON storage mechanism that `signac` provides, and we check the document to determine when the operation has been completed using the `FlowProject.post` decorator. Any function of a job can be used as a pre- or post-condition. In this case, we simply look for the `tmax` key in the job document using the `complete` function. Note the `FlowProject.label` decorator for this function; we will discuss this in further detail below.

Although this particular example is quite simple, in principle any workflow that can be represented by a directed graph may be encoded and executed using `signac-flow`. In the context of `signac-flow`, individual operations are the nodes of a graph, and the pre- or post-conditions associated with each operation determine the vertices. To simplify running such workflows, by default the `project.py` run interface demonstrated in Fig. 3 will automatically run the entire workflow for every job in the workspace. When conditions are defined in the manner shown above, `signac-flow` will ensure that only incomplete tasks are run, i.e., in this example, once `tmax` has been calculated for a particular job, the `calculate` operation will not run again for that job. Rather than running everything at once, it is also possible to exercise more fine-grained control over which operations to run using `signac-flow`:

```
$ # Runs all outstanding operations for all jobs
$ python project.py run
$ # `exec` ignores the workflow and just runs a
$ # specific job-operation
$ python project.py exec ${OP} ${JOB_ID}
$ # Run up to two operations for a specific job
$ python project.py run -j ${JOB_ID} -n 2
```

A critical feature of the `signac` framework is its scalability to HPC. The file-based data model is designed to leverage the high performance file systems common on such systems, and workflows designed locally are immediately executable on HPC clusters. In particular, any operation that can be successfully executed in the manner shown in Fig. 3 can also be immediately submitted to cluster schedulers. The `signac-flow` package achieves this by creating cluster job scripts that perform the above operations:

```
$ # Print the script for one 12-hour job
$ # Additional scheduler directives are customizable
$ python project.py submit -n 1 -w 12 --pretend
Query scheduler...
Submitting cluster job 'Projectiles/d61...':
- Operation: calculate(d61...)
#PBS -N Projectiles/d61...
#PBS -l walltime=12:00:00
#PBS -l nodes=1
#PBS -V

set -e
set -u
```

```
cd /path/to/project
```

```
# Operation 'calculate' for job 'd61...':
python project.py exec calculate d61
```

The workflow tracking functionality of `signac-flow` also extends to compute clusters. Users can always check the status of particular jobs to see how far they have progressed in the workflow, and when working on a system with a scheduler, `signac-flow` will automatically provide information about the status of jobs submitted to the scheduler. Depending on the desired verbosity, this status information can be output in a variety of formats. A relatively detailed version of the output is shown here:

```
$ # Submit 3 random jobs for 12 hours
$ python project.py submit -n 3 -w 12
$ # Status output has options to control detail
$ python project.py status -de
# Overview:
Total # of jobs: 15

label      ratio
-----
complete  |#-----| 6.67%

# Detailed View:

## Labels:
job_id      labels
-----
00e5f0c36294f0eee4a30cabb7c6046c  complete
d61ac71a00bf73a38434c884c0aa82c9
...

## Operations:
job_id  operation  eligible  cluster_status
-----
d61ac7  calculate  Y         Q
41dea8  calculate  Y         A
585599  calculate  Y         Q
2fc415  calculate  Y         I
...
```

In the overview section, we see that 6.67%, or $\frac{1}{15}$ jobs have completed, reflecting the job run locally in Fig. 3. The rows in this section are populated by any function decorated with the `FlowProject.label` decorator, with each row showing the percentage of jobs that evaluate to `True` for that function. While any callable, such as a lambda expression, could be used as a pre- or post-condition, using a function decorated in this manner makes it easy to track total progress through the workflow. The labels section below the overview provides the same information on a per-job basis, in this case showing which jobs have completed and which have not.

Finally, the operations section indicates the progress of jobs on a per-operation basis. In this particular view, the `eligible` column is redundant because we have omitted completed operations for brevity; however, if we requested a complete listing, the job marked as complete in the labels section would be listed here with an `N` in the `eligible` column. In this instance, there are fourteen jobs remaining that are eligible for the `calculate` operation, of which three have been submitted to the cluster (and are therefore marked as active). Of these three, one has actually begun running (and is marked as `[A]`), while the other two indicate that they are queued (marked as `[Q]`). The final job shown is inactive on the cluster (`[I]`) as it has not yet been submitted.

The quick overview of this section highlights the core features of the `signac` framework. Although the example demonstrated here is quite simple, the data model scales easily to thousands of


```
In [4]: %%writefile project.py
import flow
from flow.project import FlowProject

@FlowProject.label
def complete(job):
    return 'tmax' in job.document

@FlowProject.operation
@FlowProject.post(complete)
def calculate(job):
    import numpy as np
    g = 9.81
    roots = np.roots([-g/2, np.sin(job.sp.theta), 0])
    tmax = roots[roots != 0][0]
    job.doc.tmax = tmax

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

Writing project.py

```
In [5]: !python project.py run
```

Execute operation 'calculate(4f8a64741a09749ac1320f4b61292e0c)'...

```
In [6]: !find workspace
job.document()
```

```
workspace
workspace/4f8a64741a09749ac1320f4b61292e0c
workspace/4f8a64741a09749ac1320f4b61292e0c/signac_job_document.json
workspace/4f8a64741a09749ac1320f4b61292e0c/signac_statepoint.json
```

```
Out[6]: {'tmax': 0.2038735337271834}
```

Fig. 3: The `signac-flow` module enables the easy automation of workflows operating on `signac` workspaces. Here we demonstrate such a workflow operating on the data space defined in Fig. 2. In this case, the workspace consists only of one job; the real power of the `FlowProject` arises from its ability to automatically handle an arbitrary sequence of operations on a large number of jobs. Note that in this figure we are still assuming $v=1$ for simplicity.

data points and far more complex and nonlinear workflows. More involved demonstrations can be seen in the documentation¹, on the `signac` website², or in the original paper published in the *Journal of Computational Materials Science* [ADRG18].

Design and Implementation

Having provided an overview of `signac`'s functionality, we will now delve into the specifics of its implementation. The central element of the framework is the `signac` data management package, which provides the means for organizing data directly on the filesystem. The primary requirement for using this database

is that every job (data point) in the data space must be uniquely indexable by some set of key-value pairs, namely the job state point. The hash of this state point defines the job id, which in turn is used to define the directory where data associated with this job is stored. To ensure that the state point associated with the job id can be recovered, a JSON-encoded copy of the state point is stored within this directory.

This storage mechanism enables $O(1)$ access to the data associated with a particular state point through its hash as well as $O(N)$ indexing of the data space. This indexing is performed by traversing the data space and parsing the state point files directly; other files may also be parsed along the way if desired. In general, `signac` automatically caches generated indexes within a single session where possible, but for better performance after start-up

1. <http://signac.readthedocs.io>

2. <http://signac.io>

the indexes can also be stored persistently. These indexes then allow efficient selection and searching of the data space, and MongoDB-style queries can be used for complex selections.

This distributed mode of operation is well-suited to the high performance filesystems common to high performance computing. The explicit horizontal partitioning and distributed storage of data on a per-job basis is well suited to HPC operations, which are typically executed for multiple jobs in parallel. Since data is accessed distributively, there is no inherent bottleneck posed by funneling all data read and write operations through one or more server applications. Further sharding across multiple filesystems, for instance, could be accomplished by devising a scheme to divide a project's data into multiple workspaces that would then be indexed independently.

From the Python implementation standpoint, the central component to the `signac` framework is the `Project` class, which provides the interface to `signac`'s data model and features. In addition to the core index-related functionality previously mentioned, the `signac Project` also encapsulates numerous additional features, including, for example, the generation of human-readable views of the hash-obfuscated workspace; the ability to move, copy, or clone a full project; the ability to synchronize data across projects; and the detection of implicit schema. We qualify these schema as implicit because they are only defined by the state points of jobs within the workspace, *i.e.* there is nothing like a table schema to enforce a particular structure for the state points of individual jobs. Searching through or iterating over a `Project` instance generates `Job` objects, which provide Python interfaces to the jobs within the project and their associated data. In addition to providing a Pythonic access point to the job state point and the job document, a `Job` object can always be mapped to its location on the filesystem, making it ideal for associating file-based data with the appropriate data point.

The central object in the `signac-flow` package is the `FlowProject` class, which encapsulates a set of operations acting on a `signac` data space. There is a tight relationship between the `FlowProject` and the underlying data space, because operations are in general assumed to act on a per-job basis. Using the sequence of conditions associated with each operation, a `FlowProject` also tracks workflow progress on per-job basis to determine which operations to run next for a given job. Different HPC environments and cluster schedulers are represented by separate Python classes that provide the means for querying schedulers for cluster job statuses, writing out the job scripts, and constructing the submission commands. Job scripts are created using templates written in `jinja2` [Ron], making them easily customizable for the requirements of specific compute clusters or users. This means that workflows designed on one cluster can be easily ported to another, and that users can easily contribute new environment configurations that can be used by others. Currently, we support Slurm and TORQUE schedulers, along with more specialized support for the following supercomputers (listed along with their funding organizations): XSEDE Comet, XSEDE Stampede, XSEDE Bridges, INCITE Titan, INCITE Eos, and the University of Michigan Flux clusters.

The `signac` framework prioritizes modularity and interoperability over monolithic functionality, making it readily extensible. One of the tools built on top of the core infrastructure is `signac-dashboard` [Bra18], a web interface for visualizing `signac` data spaces that is currently under active development. All tools in the framework, including `signac-flow`, share the

`signac` database as a core dependency. Aside from that, however, core `signac` and `signac-flow` avoid any hard dependencies and are implemented as pure Python packages compatible with Python 2.7 and 3.3+. In conjunction with the framework's full-featured command line interface, these features of the framework ensure that it can be easily incorporated into any existing file-based workflows, even those using primarily non-Python tools.

Comparisons

In recent years, many Python tools have emerged to address issues with data provenance and workflow management in computational science. While some are very similar to the `signac` framework in their goals, a major distinction between `signac` and other comparable tools is that the `signac` data management component is independent of `signac-flow`, making it much easier to interact with the data outside the context of the workflow. As a result, while these packages solve problems similar to those addressed by `signac`, they take different and generally less modular approaches to doing so. Other packages have focused on the distinct but related need for complete provenance management for reproducibility. These tools are orthogonal to `signac` and may be used in conjunction with it.

Workflow and Provenance Management

Two of the best-known, most comparable Python workflow managers are Fireworks [JOC+15] and AiiDA [PCS+16]. Fireworks and AiiDA are full-featured workflow managers that, like `signac-flow`, interface with high performance compute clusters to execute complex, potentially nonlinear workflows. These tools in fact currently offer more powerful features than `signac-flow` for monitoring the progress of jobs, features that are supported by the use of databases on the back end. However, maintaining a server for workflow management can be cumbersome, and it introduces additional unnecessary complexities.

A more significant limitation of these other tools is that their data representations are closely tied to the workflow execution, making it much more challenging to access the data outside the context of the workflow. Concretely, these software typically store data in a specific location based on a particular instance of an operation's execution, so the data can only be found by looking for that specific instance of the operation. Conversely, in `signac` the data is identified by its own metadata, namely its state point, so once it has been generated its access is no longer linked to a specific instance of a `signac-flow` operation (assuming that `signac-flow` is being used at all).

Of course, knowing exactly where and how data was generated and transformed, *i.e.*, the data provenance, is also valuable information. Two tools that are specialized for this task are Sacred [GKC+17] and Sumatra [Dav12]. Superficially, the `signac` framework appears especially similar to Sacred. Both use decorators to convert functions into executable operations, and configurations can be injected into these functions (in `signac`'s case, using the job object). Internally, Sacred and `signac-flow` both depend on the registration of particular functions with some internal API: in `signac-flow`, functions are stored as operations within the `FlowProject`, whereas Sacred tracks functions through the `Experiment` class. However, the focus of Sacred is not to store data or execute workflows, but instead to track when an operation was executed, the configuration that was used, and what output was generated. Therefore, in principle `signac` and

Sacred are complementary pieces of software that could be used in concert to achieve different benefits.

We have found that integrating Sacred with `signac` is in fact quite simple. Once functions are registered with either a Sacred Experiment or a `signac-flow` `FlowProject`, the operations can be run either through Python or on the command line. While both tools typically advocate using their command line interfaces, the two can be integrated by using one from the command line while having it internally call the other through the corresponding Python interface. When used in concert with `signac`, the primary purpose of the Sacred command line interface, the ability to directly interact with the configuration, is instead being managed by the underlying `signac` database; in principle, the goal of this integration would be to have all configuration information tracked using `signac`. Conversely, `signac-flow`'s command line interface offers not only the ability to specify which parts of the workflow to run, but also to query status information or submit operations to a scheduler with a particular set of script options. As a result, to optimally utilize both tools, we advocate using the `signac-flow` command line functionality and encoding a Sacred Experiment within a `signac-flow` operation.

The Sumatra provenance tracking tool is an alternative to Sacred. Although it is written in Python, it is primarily designed for use as a command line utility, making it more suitable than Sacred for non Python application. However, it does provide a Python API that offers greater flexibility than the command line tool, and this is the recommended mode for integration with `signac-flow` operations.

Data Management

We have found fewer alternatives to direct usage of the `signac` data model; as mentioned previously, most currently existing software packages tightly couple their data representation with the workflow model. The closest comparison that we have found is `datreant` [DSL⁺16], which provides the means for interacting with files on the file system along with some features for finding, filtering, and grouping. There are two primary distinctions between `datreant` and `signac`: `signac` requires a unique key for each data point, and `signac` offers a tightly integrated workflow management tool. The `datreant` data model is even simpler than `signac`'s, which provides additional flexibility at the cost of `signac`'s database functionality. This difference is indicative of `datreant`'s focus on more general file management problems than the issues `signac` is designed to solve. The generality of the `datreant` data model makes integrating it into existing workflows just as easy as integrating `signac`, and the `MDSynthesis` package [Dot15] is one example of a workflow tool built around a `datreant`-managed data space. However, `MDSynthesis` is highly domain-specific and it cannot be used for other types of computational studies. Therefore, while the combination of `MDSynthesis` and `datreant` is a comparable tool to the `signac` framework in the field of molecular simulation, it does not generalize to other use-cases.

Conclusions

The `signac` framework provides all the tools required for thorough data and workflow management in scientific computing. Motivated by the need for managing the dynamic, heterogeneous data spaces characteristic in computational sciences, the tools are

tailored for the use-cases most commonly faced in this field. The framework has strived to achieve high ease of use and interoperability by emphasizing simple interfaces, minimizing external requirements, and employing open data formats like JSON. By doing so, the framework aims to minimize the initial barriers for new users, making it easy for researchers to begin using `signac` with little effort. The framework frees computational scientists from repeatedly solving common data and workflow problems throughout their research, and at a higher level, reduces the burden of data sharing and provenance tracking, both of which are critical to accelerating the production of reproducible and reusable scientific results.

Acknowledgments

We would like to thank all contributors to the development of the framework's components, J.A. Anderson, M.E. Irrgang and P. Damasceno for fruitful discussion, feedback and support, and B. Swerdlow for his contributions and feedback and coming up with the name. We would also like to thank all early adopters that provided feedback and thus helped in guiding and improving the development process. Development and deployment supported by MICCoM, as part of the Computational Materials Sciences Program funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, Materials Sciences and Engineering Division, under Subcontract No. 6F-30844. Project conceptualization and implementation supported by the National Science Foundation, Award # DMR 1409620.

REFERENCES

- [ADRG18] Carl S. Adorf, Paul M. Dodd, Vyas Ramasubramani, and Sharon C. Glotzer. Simple data and workflow management with the `signac` framework. *Computational Materials Science*, 146:220 – 229, 2018. URL: <http://www.sciencedirect.com/science/article/pii/S0927025618300429>.
- [Bra18] Bradley D. Dice. `signac-dashboard`, 2018. URL: <https://bitbucket.org/glotzer/signac-dashboard/src/master/>.
- [Cor16] Oracle Corporation. `MySQL`, 2016. URL: <https://www.mysql.com>.
- [Dav12] Andrew P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Eng.*, 14:48–56, 2012.
- [Dot15] David L. Dotson. `MDSynthesis`: a Python package enabling data-driven molecular dynamics research, July 2015.
- [DSL⁺16] David L. Dotson, Sean L. Seyler, Max Linke, Richard J. Gowers, and Oliver Beckstein. `datreant`: persistent, pythonic trees for heterogeneous data. In S Benthall and S Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 51–56, Austin, TX, 2016.
- [Ecm17] Ecma. The JSON Data Interchange Syntax, December 2017. URL: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.
- [GKC⁺17] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The sacred infrastructure for computational research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49–56, Austin, TX, 2017.
- [JOC⁺15] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. `Fireworks`: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.
- [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.
- [Mon16] MongoDB, Inc. MongoDB, 2016. URL: <https://www.mongodb.com/>.
- [PCS⁺16] Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky. AiiDA: automated interactive infrastructure and database for computational science. *Comput. Mater. Sci.*, 111:218–230, 2016.
- [PG07] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. URL: <http://ipython.org>.
- [Ron] Armin Ronacher. jinja2. Accessed on 2017/09/29. URL: <http://jinja.pocoo.org/>.

Yaksh: Facilitating Learning by Doing

Prabhu Ramachandran^{§¶*}, Prathamesh Salunke[‡], Ankit Javalkar[‡], Aditya Palaparthi[‡], Mahesh Gudi[‡], Hardik Ghaghada[‡]

<https://youtu.be/ngrfZlgrnW4>

Abstract—Yaksh is a free and open-source online evaluation platform. At its core, Yaksh focuses on problem-based learning and lets teachers create practice exercises and quizzes which are evaluated in real-time. A large array of question types like multiple choice, fill-in-the-blanks, assignment upload, and assertion or standard I/O based programming questions are available. Yaksh supports Python, C, C++, Java, Bash, and Scilab programming languages. In addition, Yaksh allows teachers to create full-blown courses with video and/or markdown text-based lessons. Yaksh is designed to be secure, easily deployable, and can scale-up to 500+ users simultaneously.

Introduction

Yaksh is created by the [FOSSEE Python team](http://fossee.in). The *FOSSEE project* (<http://fossee.in>) based at IIT Bombay, is funded by the Ministry of Human Resources and Development, MHRD (<http://mhrd.gov.in>) of the Government of India. The goal of the FOSSEE project is to increase the adoption of Free and Open Source Software in Education in India. The project started in 2009 to develop and promote a variety of open source projects. FOSSEE's Python group attempts to promote the adoption of Python in India. More details on the activities of the Python group of FOSSEE have been presented earlier at SciPy 2016 [PR2016]. Yaksh was described briefly there. However, Yaksh has evolved considerably in the last few years. It has been used for several courses at IIT Bombay as well as online. In addition, Yaksh provides a simple interface to host a MOOC and we discuss this feature as well.

As part of FOSSEE's efforts we have created learning material for Python and have conducted hundreds of workshops on Python. We find that to effectively train people to learn to program, it is imperative to make them solve programming problems. Yaksh has been created by FOSSEE for this purpose.

Overview of Yaksh

Since the emergence of learning management system (LMS) and massive open online course (MOOC) providers, e-learning has grown significantly. Despite the ever increasing adopters, major platforms still use simple questions like multiple-choice questions

and uploading of assignments from students as a means to evaluate students' performance. Yaksh seeks to improve on this.

It is well known that practice assignments and problem solving improve understanding. In the case of programming languages, this is especially so. Programming is a skill and to develop it, one must necessarily write programs. Providing an interface where users can attempt a question and immediately obtain feedback on the correctness of their program would be very useful both to a student and to a teacher. This same interface could also be used to assess the performance of the student. This is useful for the student to understand where they can improve and to the teacher to identify which concepts are not properly understood by the students. In the Indian context, a recent study [AM2017] found that even though there are many graduates with a computer science background, only 5% of the students are able to write programs with the correct logic. Indeed, our own experience is that many students learn computer science theoretically without writing too many computer programs. It is therefore important to provide a tool that facilitates practice programming and programming assessment.

In 2011, the first version of Yaksh was developed to administer programming quizzes for an online teacher training course that FOSSEE conducted. More than 600 teachers were trained and we wanted them to be able to write programs and have those corrected automatically. This work was presented at SciPy India 2011 [PR11].

Yaksh is a free and open-source online evaluation software that allows teachers to create courses and students to watch lessons and attempt tests which are evaluated immediately. Yaksh is designed to be used by a large number of users concurrently thereby making it apt for use in schools, colleges and other educational institutes for training a large number of students.

Yaksh is implemented in Python and uses Django (<https://www.djangoproject.com/>). It can be installed as a Django app with the *pip* command, thus allowing other Django based web projects to install the app within their project. The sources are available from: https://github.com/FOSSEE/online_test

To use Yaksh, one could sign-up on the official <https://yaksh.fossee.in> website or host it on one's own servers. The most standard and secure way to deploy Yaksh on a server is to build separate docker images using docker compose. Instructions for this are available in the Yaksh sources and are easy to setup.

For teachers, Yaksh provides a wide array of question types which include the basic question types like multiple choice, fill-in-the-blanks, assignment upload, etc. One can also add standard I/O and assertion-based questions for simple and basic programming

* Corresponding author: prabhu@aero.iitb.ac.in

§ Department of Aerospace Engineering

¶ IIT Bombay, Mumbai, India

‡ FOSSEE IIT Bombay, Mumbai, India

questions. For complex programs, teachers can add a hook-based test case which enable them to take the student answer and evaluate it in whatever way they want. Once the questions are created, they can create a question paper that can be added to a practice exercise or a quiz. The question paper can have a mixed set of fixed questions or a random set of questions selected from a pool of questions. In conjunction with quizzes, teachers can also add video or markdown-based lessons. Teachers can also monitor students in real-time during a test, as well as their overall progress for the course, thereby gaining insight on how students are performing.

Yaksh is designed to be easy-to-use by a student. All they have to do is sign-up, enroll for a course and start. They could go through the lessons, practice a few questions and then attempt the quiz, on which their performance is rated. While doing so, they get easy-to-understand feedback for any mistakes they make from the interface, thereby improving their answers.

Yaksh is being used extensively by the FOSSEE team to teach Python to many students all across India. Over 6000 students have used the interface to learn Python. It has been used in several courses taught at IIT Bombay and also for conducting recruitment interviews internally.

There are a few other open source software packages that do all or part of what Yaksh does.

[nbgrader](#) is a Jupyter Notebook plugin that can be used to grade programming assignments. The student submits jupyter notebooks containing code blocks which are then evaluated manually or automatically. [nbgrader](#) provides a very convenient Jupyter based interface. Instead, Yaksh offers instant feedback and grading, supports a variety of different languages, and also allows one to host a full course.

[relate](#) is similar to Yaksh in scope and goals. It allows a user to create a web based course with a grading interface quite similar to Yaksh. However, entering content into the software is based largely on YAML which is great for developers but not all end-users. Yaksh provides several question types and different ways to evaluate students' code.

[Datacamp](#) also provide several tools that are well suited for hosting very attractive courses online. It provides an easy to use and interactive interpreter for programming, which is also pluggable. However, it is not necessarily designed from the ground up for online assessment of students and live quizzes and exercise programs.

In this paper we first discuss how Yaksh may be installed, its features, and a high-level overview of its design and implementation. We then present some information on how Yaksh has been used at FOSSEE for a variety of tasks.

Installation and setup

Deployment of a web application for development or for production purposes, should be as easy as possible. There are a few different ways of setting up Yaksh:

- Trial instance with Docker
- Trial instance without Docker
- Production instance using Docker and Docker compose.

Yaksh can be deployed with a limited number of commands using the [invoke](#) Python package to make the deployment as easy as possible.

Yaksh is written in Python and depends on Django and a few other Python dependencies. The dependencies can be installed

using the [pip](#) package manager tool. It is recommended to use Yaksh along with Docker.

Yaksh can be cloned from the Github repository. To do this one can run:

```
$ git clone https://github.com/FOSSEE/online_test.git
$ cd online_test
```

One can then install the required dependencies, for Python 2, by running:

```
$ pip install -r requirements/requirements-py2.txt
```

or for Python 3, by running:

```
$ pip install -r requirements/requirements-py3.txt
```

It is recommended that one use Python 3 to run Yaksh.

Quickstart

The method discussed here allows a user to setup a local instance of Yaksh to try the platform for a limited number of users. Yaksh can be run within a demo instance on a local system to try the platform for a limited number of users. To set up a demo instance one can run:

```
$ invoke start
```

This command will start the code server within a docker environment.

In case docker is not available, the code server can also be run without docker by running:

```
$ invoke start --unsafe
```

However, this is not recommended since this leaves the base system potentially vulnerable to malicious code. In case one wishes to use this method, all Python dependencies will have to be installed using `sudo`.

In order to access the interface, one can run the web server using:

```
$ invoke serve
```

This command will run the Django application server on the **8000** port and can be accessed using a browser.

Production Setup With Docker

In order to setup Yaksh on a Production server with docker compose, one first needs to set certain environment variables. To do so, one can create a `.env` file with the following details:

```
DB_ENGINE=mysql
DB_NAME=yaksh
DB_USER=root
DB_PASSWORD=db_password
DB_PORT=3306
```

The local system needs to have [Docker Compose](#) installed. Then, one must navigate to the Docker directory:

```
$ cd /path/to/online_test/docker
```

Running the following commands will ensure that the platform is setup:

```
$ invoke build
```

```
$ invoke begin
$ invoke deploy --fixtures
```

The `build` command builds the docker images, the `begin` command spawns the docker containers and the `deploy` command runs the necessary migrations.

The demo course/exams

Since setting up a complete course with associated Modules, Lessons, Quizzes and Questions can be a tedious process for a first time user, Yaksh allows moderators to create a Demo Course by clicking on the 'Create Demo Course' button available on the dashboard.

One can then click on the Courses tab and browse through the Demo Course that has been just created.

One can read more about Courses, Modules, Lessons and Quizzes in the sections below.

Basic features of Yaksh

Once Yaksh is installed and running, one can create a full fledged course with lessons, practice, and evaluation based quizzes. Yaksh supports following languages such as Python, Java, C, C++, and Scilab. It provides several question types such as Single Correct Choice (MCQ), Multiple Correct Choice (MCC), Programming, Fill in the blanks, Arrange the options, Assignment upload. For simple and complex questions several test case types are provided such as standard input/output test case, Standard Assertion test case, Hook based test case, MCQ based test case, etc. The interface provides instant feedback for the student to improve their submissions. While administering quizzes or practice sessions, one can monitor the student's progress in real-time. This is particularly useful in practice sessions so as to help students who are not doing well. Finally, a student gets a certificate after successful completion of a course.

All the features are explained in detail in the workflow section.

Internal design

The two essential pieces of Yaksh are:

- Django Server
- Code server

Fig 1 shows the workflow for the evaluation of code submitted by a student and how this relates to these two pieces.

Django Server

Django is a high-level Python web framework. Django makes it is easy to create web applications, handles basic security issues, and provides a basic authentication system.

Django makes it easy to store information in a database by providing an object-relational mapping (ORM). This allows users to define the database tables at a very high level without having to write raw SQL queries.

Django provides a view controller to handle the requests sent from the client side. A view then interacts with the database using the ORM, retrieves data and pushes it to a template for rendering into an HTML page.

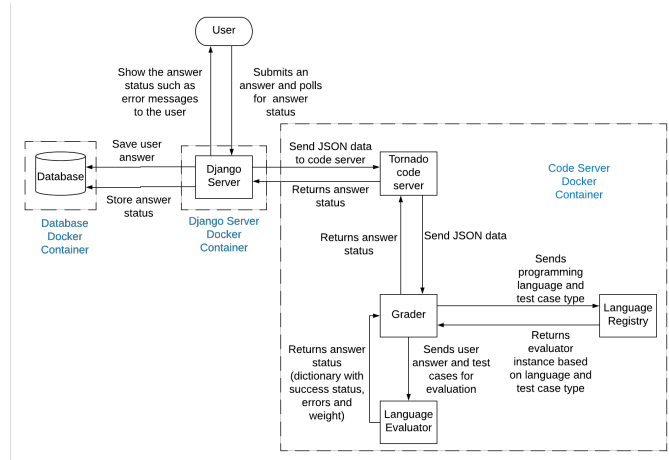


Fig. 1: Flow diagram for code evaluation procedure

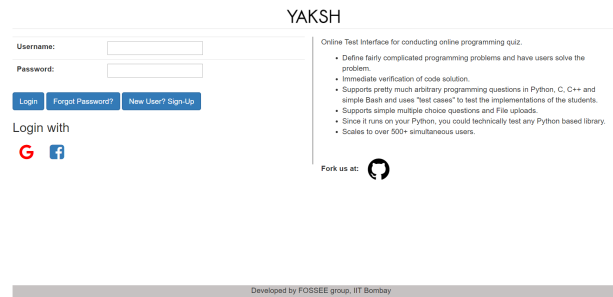


Fig. 2: The Yaksh application login screen

Authentication system

Yaksh uses the Django authentication system for handling basic user authentication, cookie-based user sessions and permissions for users and groups. Additionally, Yaksh uses email verification to provide users with a second layer of security while creating user accounts. To create an account on Yaksh, one can either go to the website and sign-up or can sign-up via the OAuth system provided for Google and Facebook accounts. By default the user is logged-in as a **student**, although the user can become a moderator if the user is added to the **moderator** group. Fig. 2 shows the login screen for Yaksh.

Yaksh models

A Django model is a Python class that subclasses `django.db.models.Model` representing the database table. Each attribute of the model represents a database table field.

The models for Yaksh are as follows:

- User: This is the default model provided by Django for storing the user name, first name, last name, password etc.
- Profile: This model is used for storing more information about a user such as institute, department etc.
- Question: This model is used for storing question information such as name, description etc. Once the questions are created they are added in the question paper
- TestCase

This model is used for storing question test cases.

Different test case models are available which subclass the `TestCase` model. Some of these are -

- `StandardTestCase`
This model is used for test cases that use assertions to test success or failure.
 - `StdIOBasedTestCase`
This model is used for test cases based on the standard output produced by a test.
 - `McqTestCase`
This model is used for MCQ (single correct choice) or MCC (multiple correct choice) type of question.
 - `HookTestCase`
This model is used for questions where there is a need for more complex testing. This model comes with a predefined function `check_answer` where the student answer (path to user submitted files for assignment uploads) is passed as an argument. The question creator can hence scrutinise the user answer in much more specific ways.
 - `Fill in the blanks Test case`
This model supports integer, float, string types for fill in the blanks questions.
 - `ArrangeTestCase`
This model is used for creating a test case with jumbled options which can be re-ordered by students.
- `Course`
Is used for creating a course.
 - `Quiz`
Is used for creating a quiz.
 - `QuestionPaper`
Is used for creating a questionpaper for a quiz containing all the questions for the quiz.
 - `AnswerPaper`
Is used for storing the answer paper for a particular course and quiz.
 - `Answer`
Is used for storing the answer submitted by the user which are added to the answer paper.
 - `Lesson`
A lesson can be any markdown text which can have an embedded video of a particular topic.
 - `LearningUnit`
A learning unit can either be a lesson or a quiz.
 - `LearningModule`
A learning module can be any markdown text which can have an embedded video of a particular topic. A learning module contains learning units.

Code Server

The Code Server is an important part of Yaksh. The evaluation of any code is done through the code server. We have used the [Tornado](#) web framework to manage the asynchronous process generation. A `settings.py` file is provided which is used to specify various parameters necessary for the code server.

This settings file contains information such as:

- number of code server processes required to process the code.

```
code_evaluators = {
    "python": {"standardtestcase": "yaksh.python_assertion_evaluator.PythonAssertionEvaluator",
              "stdiobasedtestcase": "yaksh.python_stdio_evaluator.PythonStdIOEvaluator",
              "hooktestcase": "yaksh.hook_evaluator.HookEvaluator"
            },
}
```

Fig. 3: Dictionary mapping of Python code evaluator

- server pool port, a common port for accessing the Tornado web server.
- server host name, a server host for accessing the Tornado web server.
- a timeout to prevent infinite loops locking up a process.
- dictionary of code evaluators based on the programming language.

A Tornado HTTP server is started with the specified server hostname and pool port from the settings. The server takes the following arguments -

- `UID of an answer`: This is the unique ID associated with an answer submitted. This is specifically required to poll the server for the status of the submitted answer.
- `JSON Data`: This contains all the data required for evaluation of a code answer, namely, user answer, language of the question, test cases associated with the question, and files required by the code, if any.
- `User directory`: Every user is allotted a user directory, in which script files are executed. The path of this user directory is passed to the server.

The aforementioned arguments are passed to the Tornado server which takes the JSON data and sends it to a `Grader` for unpacking. The `Grader` unpacks the data, selects a language evaluator using a language registry and sends it to that language evaluator for evaluation. The language evaluator takes the user answer and evaluates it in the specified user directory. The evaluator then sends the output of the evaluation back to the Tornado server through the `Grader`. The Django server, meanwhile, keeps polling the Tornado server for the status of the evaluation. If the evaluation is complete, the Tornado server hands over the data to the Django server for saving and displaying.

Grader

Grader extracts the data such as language, test case type, test cases, user directory path from json metadata sent to it. It then creates the user directory from the path. Then it sends the test case type and language information to the language registry to get the evaluator. Once the evaluator is obtained, grader calls the evaluator and sends the test cases, user answer to the evaluator and code execution starts.

Language Registry

The language registry takes a programming language and test case type and generates a evaluator instance using the dictionary mapping in the settings file and returns the evaluator instance to the Grader.

Dictionary mapping of evaluator is as shown in Fig 3

For example say `Python` language and `standard assert` test case type are set during question creation, then Python assertion evaluator is instantiated from the dictionary mapping and the created instance is returned to grader.

Evaluators

Evaluators are selected based on the programming language and test case type set during the question creation.

For each programming language and test case type separate evaluator classes are available.

Each evaluator class subclasses `BaseEvaluator`. The `BaseEvaluator` class includes common functionality such as running a command using a Python subprocess, creating a file, and writing user code in the file, making a file executable etc.

There are several important aspects handled during code evaluation:

- **Sandboxing**
A user answer might be malicious i.e. it might contain instructions which can access the system information and can damage the system. To avoid such a situation, all the code server process run as "nobody" so as to minimize the damage due to malicious code.
- **Handling infinite loops**
There are chances that user answers contain infinite loops and lock up a process. To avoid this, code is executed within a specific time limit. If the code execution is not finished in the specified time, a signal is triggered to stop the code execution and sending a message to the user that code might contain an infinite loop. We use the `signal` module to trigger the `SIGALARM` with the server timeout value. Unfortunately, this does make our code server Linux/MacOS specific.
- **Docker**
To make the code evaluation more secure all the code evaluation is done inside a docker container. Docker containers can also be used to limit the use of system resources such as CPU utilization, memory utilization etc.

Workflow of Yaksh

Instructor workflow

An instructor (also called the moderator) has to first create a course before creating a quiz, lesson or module. Before creating a quiz, the instructor has to create some questions which can be added to a quiz. The instructor can create any number of questions through the online interface. These can be either multiple-choice, programming, assignment upload, fill in the blanks or arrange option questions. All these question types must be accompanied with several test cases. A sample Python question along with its test case is shown in the Fig. 4 and Fig. 5. The instructor can set minimum time for a question if it is part of an exercise. A question can have partial grading which depends on a weightage assigned to each test case. A question can have a solution which can be either a video or any code. This allows us to pose a question, ask the student to attempt it for a while and then show a solution.

A programming question can have test case types of standard assert, standard I/O or a hook. Fig. 5 shows a sample test case of standard assert type. In a similar way, the instructor can add test cases for standard I/O. For simple questions, standard assert and standard I/O type test cases can be used. For complex questions, hook based test case is provided where the user answer is provided to the hook code as a string and instructor can write some code to check the user answer. For other languages assertions are not easily possible but standard input/output based questions are easy to create. The moderator can also create a question with jumbled options and student has to arrange the options in correct order.

The screenshot shows the 'Question interface' in the Yaksh system. It includes the following fields and content:

- Summary:** Quiz1: Find number of u
- Language:** Python
- Type:** Code
- Points:** 1
- Rendered:** Write a function called `count_unique(text)` to find the number of unique words in a given string. Ignore case and assume no punctuation is in the words. For example:


```
>>> count_unique('a banana He HE he')
3
>>> count_unique('ha ha ha')
1
```
- Description:** Write a function called `<code>count_unique(text)</code>` to find the number of unique words in a given string. Ignore case and assume no punctuation is in the words. For example:


```
<br>
<br>
<code>count_unique('a banana He HE he')</code>
```
- Tags:** quiz1
- Rendered Solution:** Solution: (Empty text area)
- Snippet:** Snippet: (Empty text area)
- Minimum Time (in minutes):** 0
- Partial Grading:**
- Grade Assignment Upload:**
- File:** Browse... No files selected.

Fig. 4: Question interface

The screenshot shows the 'Sample Test case' interface. It includes the following fields and content:

- Type:** Standard Testcase
- Test case:** `assert count_unique('a aaa a aba') == 2`
- Weight:** 1
- Command Line arguments for bash only:** (Empty text area)
- Test case args:** (Empty text area)
- Delete:**

Fig. 5: Sample Test case

Detailed instructions on creating a question and test cases are provided at (<https://yaksh.readthedocs.io>).

The moderator can also import and export questions. The moderator then creates a quiz and an associated question paper. A quiz can have a passing criterion. Quizzes have active durations and each question paper will have a particular time within which it must be completed. For example, one could conduct a 15 minute quiz with a 30 minute activity window. Questions are automatically graded. A user either gets the full marks or zero if the tests fail. If a question is allowed to have partial grading then during evaluation the user gets partial marks based on passing test cases.

The moderator can then create learning modules. A module encapsulates learning units, i.e., lessons and quizzes. A lesson can have description either as a markdown text or a video or both. After lesson creation, the moderator can create modules. A module can have its own description either as a markdown text or a

Course Name: Basic Programming Using Python(01 - 30Apr)
 Quiz Name: Quiz-1 (Self Learning)
 Number of papers: 168
 Papers completed: 162
 Papers in progress: 16
 Question Statistics

Download CSV

Name	Username	Roll number	Institute	Questions answered	Marks obtained	Attempts	Time Remaining	Status
Kuldeep Verma	kuldeep3	01515003116	MSIT	10.0	10	0.0.0	completed	
Yogeshwar Bari	yogeshwarbari	07	K.K.Wagh Institute Of Engineering Education And Research Nashik	10.0	10	0.0.0	completed	
Anil Chinchawade	anilchinchawade	100	NK Orchid College of Engg	10.0	10	0.0.0	completed	
Uthayya Aich	uthayya	10900114121	Nelajai Subhash Engineering College	10.0	10	0.0.0	completed	
Varsha Mhaske	varsha_mhaske	111	SVPM's College of Engineering	10.0	10	0.0.0	completed	
Taj Kumar	tej_kumar	111416104026	prathyusha engineering college	10.0	10	0.0.0	completed	
Pritya Agrawal	divyapriya375	11268	Dr. B. C. Engineering College, Durgapur	10.0	10	0.0.0	completed	
Ajex Cruz	cruzajex	11588		10.0	10	0.0.0	completed	
Raj Patel	rapatel199871	11861		10.0	10	0.0.0	completed	
Gaurav Rai	grai	1502913040	KIET GROUP OF INSTITUTIONS	10.0	10	0.0.0	completed	
Tariq Ahmad	tariq	1504220	KIIT	10.0	10	0.0.0	completed	

Fig. 6: The moderator interface for monitoring a quiz on Yaksh.

Introduction : Preliminaries

Arun Kp with Roll no. 123 is logged in as arun_kp_student
 Developed by FOSSEE group, IIT Bombay

Fig. 7: The interface showing a video lesson

video or both. All the lessons and quizzes are added to the created module. The moderator can create any number of modules, lessons and quizzes as desired. These modules are added to a course.

Fig. 6 shows a monitor page for a quiz from one of the courses running on Yaksh. The instructors can also monitor students in real time during a quiz thereby gaining insight on how students are performing. The moderator can also view student progress for overall course, such as the number and percent of completed modules.

The moderator can regrade answerpapers using three ways:

- Answer paper can be regraded per quiz.
- Answer paper can be regraded per student.
- Answer paper can be regraded per question.

Student workflow

Working on the student side is relatively easy. After login, a student can view all the open courses or search for a course. Once the course is available, the student can enroll in a course. A student has to complete the course within a specified time. After enrolling, the student will be able to see all the modules and its units (Lessons/Quizzes) for the course. A student can view all the lessons and once the lessons are finished student can attempt the quiz. Fig. 7 shows a video lesson from the monthly running Python course.

Fig. 8 shows a MCQ question from a quiz. A student can select any one of the option and submit the answer.

Fig. 9 shows a programming question from a quiz in Python course. Once the student clicks on check answer, the answer is

Fig. 8: The interface for a multiple-choice question on Yaksh.

Fig. 9: The interface for a programming question on Yaksh.

sent to the code server for evaluation and the result from the code server is shown. From the Fig. 9 we can see that there is an indentation error in the code. Once the answer is submitted we get an indentation error message as shown in the Fig. 10. After submitting the answer, if the answer does not pass the test case then student gets an assertion error as shown in the Fig 11.

Fig. 12 shows an StdIO based question. Once the answer is submitted we get the error output as shown in Fig 13. Fig 13 shows the user output and expected output separately, indicating

Error No. 1

The following error took place:

Exception Name:	IndentationError
Exception Message:	expected an indented block (<string>, line 2)
Full Traceback:	Traceback (most recent call last): File "<string>", line 2 return n*n IndentationError: expected an indented block

Fig. 10: Error output after submitting the code answer.

Error No. 1

We tried your code with the following test case:

```
assert num_square(2) == 4
```

The following error took place:

Exception Name:	AssertionError
Exception Message:	Expected answer from the test case did not match the output

Fig. 11: Assertion Error output after submitting the code answer.

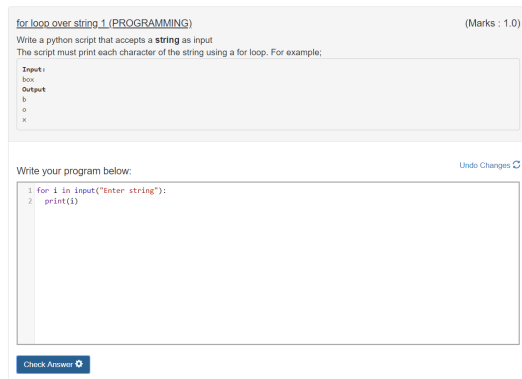


Fig. 12: The interface for a stdio question type on Yaksh.

Error No. 1

For given Input value(s): string

Line No.	Expected Output	User output	Status
1	s	Enter strings	✘
2	t	t	✔
3	r	r	✔
4	i	i	✔
5	n	n	✔
6	g	g	✔

Error: Incorrect Answer: Line number(s) 1 did not match.

Fig. 13: Error output for stdio question type.

the line by line difference between user output and expected output making it easy to trace where the error occurred.

Students can submit the answer multiple times, thereby improving their answers. Suppose a student is not able to solve a question, that question can be skipped and attempted later. All the submitted and skipped question's answers are stored so that the instructor can view all the attempts made by the student. Students can view the answerpaper for a quiz after completion.

Students can take the practice exercises where each question in the exercise is timed. Students must solve the question within the specified time, if not done within time then the solution for the question is shown and student can submit the answer once again. This makes it easy for the student to understand the mistake and correct it. These exercises run for infinite time and allows multiple attempts.

Once the course is completed, the student can view the course grades and download the certificate for that course which is generated automatically.

Supporting a new language

Adding a new language is relatively easy. In the settings file one needs to add a mapping for the evaluator corresponding to the language. An example for adding new evaluator is shown in Fig 14.

In the given Fig 14, Python is the programming language, standardtestcase, stdiobasedtestcase, hooktestcase are the test case type which are mapped to corresponding evaluator class. Here the values of the dictionary correspond to the full name of the Evaluator subclass, in this case `PythonAssertionEvaluator` is the class which is responsible to evaluate the code.

```
code_evaluators = {
    "python": {"standardtestcase": "yaksh.python_assertion_evaluator.PythonAssertionEvaluator",
              "stdiobasedtestcase": "yaksh.python_stdio_evaluator.PythonStdIOEvaluator",
              "hooktestcase": "yaksh.hook_evaluator.HookEvaluator"},
    "new_language": {
        "standardtestcase": "yaksh.new_language_assertion_evaluator.new_languageAssertionEvaluator",
        "stdiobasedtestcase": "yaksh.new_language_stdio_evaluator.new_languageStdIOEvaluator",
        "hooktestcase": "yaksh.hook_evaluator.HookEvaluator"}
}
```

Fig. 14: Dictionary mapping for new code evaluator

Separate evaluator files needs to be created for all the test case types except the hook test case.

An evaluator class should define four methods `__init__`, `teardown`, `compile_code`, and `check_code`.

- `__init__` method is used to extract all the metadata such as user answer, test cases, files (if any for file based questions), weightage (float value), `partial_grading` (boolean value).
- The `teardown` method is used to delete all the files that are not relevant once the execution is done.
- All the code compilation tasks will be performed by the `compile_code` method. There is no need to add this method if there is no compilation procedure.
- The execution of the code is performed in the `check_code` method.

The `check_code` method must return three values -

- `success (bool)` - indicating if code was executed successfully and the student answer is correct
- `weight (float)` - indicating total weightage of all successful test cases
- `error (str)` - error message if success is false

Some experiences using Yaksh

During its inception in 2011, Yaksh was designed as an evaluation interface with the idea that anyone can use Yaksh to test and grade the programming skills of students. As an evaluation interface, Yaksh was first used to evaluate 600 teachers. Since then, Yaksh has been used for teaching students, especially for courses at IIT Bombay and for conducting employment hiring tests within FOSSEE. With the introduction of Python Workshops (<https://python-workshops.fossee.in/>), an initiative of FOSSEE to remotely train students and teachers across India, Yaksh has since been refactored around the MOOC ideology, introducing the ability to learn with an emphasis on hands-on programming. We look at the various activities where Yaksh is used below.

Courses at IIT Bombay

Yaksh has been used as a online learning and testing tool for some courses at IIT Bombay. Yaksh is used to teach Python to some undergraduate students. These courses have served as a test-bed for the software. Thus far, about 300 students from IIT Bombay have been taught using Yaksh.

Usage for Python Workshops

In early 2017, FOSSEE started conducting remote Python workshops in technical colleges across India. These workshops consist of several sessions spread through one or three days, depending on the type of the course an institute chooses. A session typically

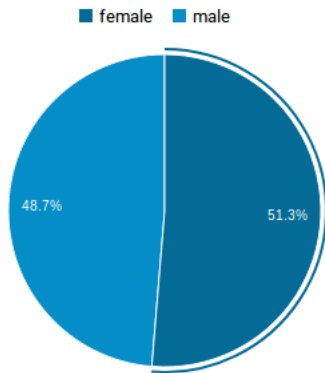


Fig. 15: Male:Female ratio of visitors on Yaksh.

begins with screening a video tutorial inside the venue. The tutorials are followed by a demanding set of exercises and quizzes, both of which are conducted on Yaksh. This is followed by brief Q&A sessions with the remote Python instructors from FOSSEE. Finally a certificate is awarded to those students who successfully finish the course. Apart from this, Yaksh also hosts a monthly, self learning online course, consisting of the same workshop materials and some bonus contents. Here are some statistics based on these activities -

- 1) As of mid 2018, around 13,000 active users are on Yaksh, with more expected to join by the end of the year.
- 2) Rapidly growing user base with about 730, 4500 and 7500 user registrations for year 2016, 2017 and mid-2018 respectively.
- 3) 100+ institutes have conducted the workshop with about 6000 students participating and about 3600 students obtaining a certificate.
- 4) For the first three months of the Python self learning course, an estimate of 3500 students enrolled with 1200 completing the course within the time frame and 400 students obtaining a passing certificate.
- 5) An equal ratio of male to female participants with most users from the age of 18-24 as seen in the Figures. 15 and 16.
- 6) Average time spent on the website by a user is around 30 minutes.
- 7) Major users are from tier 1 cities of India, regarded as highly developed IT hubs like Hyderabad, Bengaluru, Pune, and Mumbai.

Usage for hiring

One surprising use case for Yaksh has been as a tool for evaluating employment candidates by conducting tests. Yaksh has been used several times for hiring for teams functioning inside the FOSSEE project.

Plans

The team behind Yaksh is devoted to further improving user experience for both moderators and students. This includes addition of features like Instant Messaging (IM) service for moderators and teachers to guide and solve students' doubts in real time. The team also plans to add support for more programming languages to include a larger question base. Moderators will have facility to do detailed analysis on student performance in future.

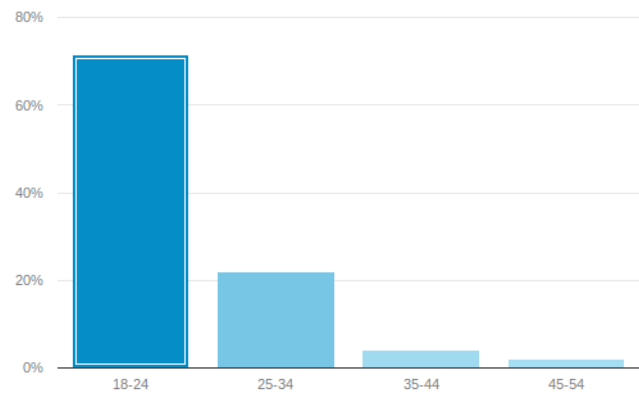


Fig. 16: Age breakdown of visitors on Yaksh.

Many colleges and schools in India do not have good internet access. We are hoping to make it easy for such institutions to locally host Yaksh using a bootable USB drive.

In addition, we are planning to make it easy for students to download the course materials and any videos in order to view the lectures offline.

For moderators, a stable web-API is being designed for other websites to harness the power of Yaksh. With this API, moderators could be able to embed lessons and quizzes available on Yaksh in Jupyter notebooks.

Conclusions

As discussed in this paper, Yaksh is a free and open source tool can be used effectively and extensively for testing programming skills of students. The features provided by Yaksh facilitates teachers to automate evaluation of students in almost real time, thereby reducing the grunt work. With addition of MOOC like features, students can learn, practice and test their programming abilities within the same place. The Python team at FOSSEE continues to promote and spread Python throughout India using Yaksh.

Acknowledgments

FOSSEE would not exist but for the continued support of MHRD and we are grateful to them for this. This work would not be possible without the efforts of the many FOSSEE staff members. The past and present members of the project are listed here: <http://python.fossee.in/about/> the authors wish to thank them all.

REFERENCES

- [PR2016] Prabhu Ramachandran, Spreading the Adoption of Python in India: the FOSSEE Python Project", Proceedings of the 15th Python in Science Conference (SciPy 2016), July 6-12, 2016, Austin, Texas, USA. http://conference.scipy.org/proceedings/scipy2016/prabhu_ramachandran_fossee.html
- [kmm14] Kannan Moudgalya, Campaign for IT literacy through FOSS and Spoken Tutorials, Proceedings of the 13th Python in Science Conference, SciPy, July 2014.
- [FOSSEE-Python] FOSSEE Python group website. <http://python.fossee.in>, last seen on May 7 2018.
- [PR11] Prabhu Ramachandran. FOSSEE: Python and Education, Python for science and education, Scipy India 2011, 4th-11th December 2011, Mumbai India.
- [AM2017] 95% engineers in India unfit for software development jobs, claims report. <http://www.aspiringminds.com/automata-national-programming-report>