DATA SCIENCE IN SCANNING PROBE MICROSCOPY:
ADVANCED ANALYTICS AND MACHINE LEARNING

A Dissertation in
Physics
by
William Dusch

© 2019 William Dusch

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2019
The dissertation of William Dusch was reviewed and approved* by the following:

Eric W. Hudson
Associate Professor of Physics
Associate Head for Diversity & Equity
Dissertation Advisor
Chair of Committee

Jorge Sofo
Professor of Physics
Professor of Materials Science & Engineering

Mauricio Terrones
Distinguished Professor of Physics
Distinguished Professor of Chemistry
Distinguished Professor of Material Science & Engineering

Roman Engel-Herbert
Associate Professor of Material Science & Engineering
Associate Professor of Chemistry
Associate Professor of Physics

Nitin Samarth
Professor of Physics
George A. and Margaret M. Downsbrough Department Head
Head of the Department of Physics

*Signatures are on file in the Graduate School
ABSTRACT

Scanning probe microscopy (SPM) has allowed researchers to measure materials’ structural and functional properties, such as atomic displacements and electronic properties at the nanoscale. Over the past decade, great leaps in the ability to acquire large, high resolution datasets have opened up the possibility of even deeper insights into materials. Unfortunately, these large datasets pose a problem for traditional analysis techniques (and software), necessitating the development of new techniques in order to better understand this new wealth of data.

Fortunately, these developments are paralleled by the general rise of big data and the development of machine learning techniques that can help us discover and automate the process of extracting useful information from this data. My thesis research has focused on bringing these techniques to all aspects of SPM usage, from data collection through analysis. In this dissertation I present results from three of these efforts: the improvement of a vibration cancellation system developed in our group via the introduction of machine learning, the classification of SPM images using machine vision, and the creation of a new data analysis software package tailored for large, multidimensional datasets which is highly customizable and eases performance of complex analyses.

Each of these results stand on their own in terms of scientific impact – for example, the machine learning approach discussed here enables a roughly factor of two to three improvement over our already uniquely successful vibration cancellation system. However, together they represent something more – a push to bring machine learning techniques into the field of SPM research, where previously only a handful of research groups have reported any attempts, and where all efforts to date have focused on analysis, rather than collection, of data. These results also represent first steps in the development of a “driverless SPM” where the SPM could, on its own, identify, collect, and begin analysis of scientifically important data.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................... vii

LIST OF TABLES ............................................................................................................. ix

ACKNOWLEDGEMENTS ................................................................................................. x

Chapter 1 Introduction ................................................................................................. 1

1.1: Motivation................................................................................................................ 1
  1.1.1: Data Science in Condensed Matter Physics ....................................................... 1
  1.1.2: Scanning Probe Microscopy ........................................................................... 2
  1.1.3: Data Science in Scanning Probe Microscopy .................................................. 2
1.2: Approach ................................................................................................................ 3
1.3: Structure of Dissertation ...................................................................................... 4

Chapter 2 Background ................................................................................................. 5

2.1: Scanning Tunneling Microscopy ......................................................................... 5
  2.1.1: Theory of Scanning Tunneling Microscopy ...................................................... 5
  2.1.2: Measurement Types .................................................................................... 7
2.2: Introduction to Machine Learning ....................................................................... 10
  2.2.1: Tasks ........................................................................................................ 11
  2.2.2: Performance Metrics .................................................................................. 11
  2.2.3: Generalization .......................................................................................... 14
2.3: Unsupervised Learning in Scanning Probe Microscopy .................................... 15
  2.3.1: Principal Component Analysis ................................................................... 15
  2.3.2: Spectral Unmixing ..................................................................................... 18
  2.3.3: Clustering .................................................................................................. 19
2.4: Deep Learning ....................................................................................................... 21
  2.4.1: Dense Neural Networks .............................................................................. 22
  2.4.2: Convolutional Neural Networks .................................................................. 23
  2.4.3: Recurrent Neural Networks ........................................................................ 25
  2.4.4: Deep Learning Process ............................................................................... 26

Chapter 3 Vibration Cancellation in Scanning Probe Microscopy using Deep Learning... 29

3.1: Motivation ............................................................................................................. 29
3.2: Basic Experimental Setup of ANITA ................................................................. 30
3.3: Linear Transfer Function Model ........................................................................ 31
3.4: Recurrent Neural Network Model ....................................................................... 31
3.5: Exploratory Analysis of Time Series .................................................................. 32
  3.5.1: Time Domain Analysis ................................................................................ 32
  3.5.2: Frequency Domain Analysis ...................................................................... 33
  3.5.3: Cross-Correlation Analysis ......................................................................... 35
3.6: Comparative Results of Predictive Models ......................................................... 36
3.7: Summary ............................................................................................................... 37
Chapter 4 Classifying Scanning Probe Microscopy Topographies using Deep Learning ......39

4.1: Motivation .................................................................................................................39
4.2: Data Collection and Annotation ..............................................................................40
4.3: Deep Learning Model ...............................................................................................43
  4.3.1: Architecture ........................................................................................................43
  4.3.2: Training Process and Hyperparameter Tuning .....................................................44
4.4: Classification Results ...............................................................................................46
4.5: Summary ....................................................................................................................48

Chapter 5 DataView: Advanced Analytics Software for Multidimensional Data ..........50

5.1: Motivation & Existing Packages ..............................................................................50
5.2: History ......................................................................................................................51
5.3: Design Highlights ......................................................................................................52
  5.3.1: History ................................................................................................................52
  5.3.2: Data Generalization .............................................................................................52
  5.3.3: Data Selectors and Viewers ................................................................................53
5.4: Summary ....................................................................................................................55

Appendix A Correlation Functions ................................................................................56

  A.1 Pearson’s correlation coefficient .............................................................................56
  A.2 Spearman’s rank correlation coefficient .................................................................57
  A.3 Polychoric (latent) correlation coefficient .............................................................57
  A.4 Cross-Spectrum Analysis .......................................................................................57

Appendix B Size of Scanning Tunneling Microscopy Data ............................................59

  B.1 Data Dimensions ......................................................................................................59
  B.2 Data Sizes ................................................................................................................59

Appendix C Programming a Convolutional Neural Network in Python .....................61

Appendix D Review of Scanning Probe Microscopy Analysis Packages ......................63

Appendix E: DataView Programmer’s Guide .................................................................65

  E.1: Delving into DataView ............................................................................................65
    E.1.1: How to install DataView ...................................................................................65
    E.1.2: Anaconda ..........................................................................................................67
    E.1.3: NumPy ..............................................................................................................67
    E.1.4: SciPy ................................................................................................................68
    E.1.5: H5Py .................................................................................................................68
    E.1.6: Pint ....................................................................................................................68
    E.1.7: Matplotlib ........................................................................................................69
    E.1.8: PyQt ................................................................................................................69
  E.2: Subpackages ............................................................................................................69
    E.2.1: data ..................................................................................................................70
    E.2.2: database ...........................................................................................................70
    E.2.3: filehandlers ..................................................................................................71
LIST OF FIGURES

Figure 2-1: Tunneling from sample to tip.................................................................6
Figure 2-2: Schematic Diagram of a Scanning Tunneling Microscope.......................8
Figure 2-3: Machine Learning: A new programming paradigm..................................10
Figure 2-4: Accuracy, Precision and Recall for binary classification........................13
Figure 2-5: Overfitting .........................................................................................14
Figure 2-6: PCA of a dataset with a multivariate Gaussian distribution......................16
Figure 2-7: PCA visualizations of a BSCCO DOS map............................................17
Figure 2-8: Linear Unmixing of a BSCCO DOS map into three endmembers. ..............19
Figure 2-9: Results of clustering of a dataset..........................................................19
Figure 2-10: Cluster visualization of the same BSCCO DOS map as Figure 2-6 ..........21
Figure 2-11: Supervised Deep Learning Framework...............................................22
Figure 2-12: Dense Neural Network........................................................................23
Figure 2-13: Typical Convolutional Neural Network Architecture..........................24
Figure 2-14: Structure of a Recurrent Neural Network ...........................................25
Figure 2-15: Deep Learning Process........................................................................26
Figure 3-1: ANITA schematic and concept..............................................................30
Figure 3-2: Model of the RNN used to predict $ZFB$ from $G$ ..................................31
Figure 3-3: Long term trend of the $Z$ signal, before and after filtering......................33
Figure 3-4: Short term trend of $Z$ an $G$ signals .................................................33
Figure 3-5: Spectrograms of $Z$ and $G$ signals.......................................................34
Figure 3-6: Global spectral densities of $G$ and $Z$ signals........................................35
Figure 3-7: Cross-Spectrum and Coherence between the $Z$ and $G$ signals .............35
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3-8: Model Performance of the LTF and RNN models</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3-9: Spectral density comparison</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4-1: Examples of STM topographies</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4-2: Spearman Correlation of Metadata</td>
<td>42</td>
</tr>
<tr>
<td>Figure 4-3: Deep learning model architecture</td>
<td>43</td>
</tr>
<tr>
<td>Figure 4-4: Training process for the deep learning model</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4-5: Hyperparameter tuning</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4-6: Model Confidence</td>
<td>48</td>
</tr>
<tr>
<td>Figure 5-1: Example of an Image Viewer</td>
<td>54</td>
</tr>
<tr>
<td>Figure 5-2: Example of a Plot Viewer</td>
<td>54</td>
</tr>
<tr>
<td>Figure E-1: Data Flow of DataView</td>
<td>74</td>
</tr>
<tr>
<td>Figure E-2: Data Structures of DataView and how they are interconnected</td>
<td>75</td>
</tr>
<tr>
<td>Figure E-3: DVMenu Example</td>
<td>91</td>
</tr>
<tr>
<td>Figure E-4: Log in Screen of DataView</td>
<td>94</td>
</tr>
<tr>
<td>Figure E-5: GUI Elements of DataView</td>
<td>98</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3-1: Mean Squared Error of Models (pm²) .........................................................36

Table 4-1: Annotations of our STM Topography Dataset. Left section is atomic quality
vs. resolution; right section is the number of images per type of material. There were
a total of 4542 images after dividing and cropping the scans. .....................................41

Table 4-2: Confusion Matrices for the final model..........................................................46

Table 4-3: Performance metrics of the classification model..........................................47

Table B-1: Common STM Dataset Sizes (double precision floating-point) .....................60

Table D-1: Scanning Probe Microscopy Packages ..........................................................63
ACKNOWLEDGEMENTS

My mother, father and brother, Judith, Raymond, and Matthew Dusch, have provided critical support for my life as a graduate student. They have provided me with the emotional support and the skills needed for me to thrive here. I would have not had the drive to complete my doctoral candidacy here without their support. I would also like to thank my girlfriend, Sarah Fry for the immense support she’s given me in my last year of graduate school.

I’ve made numerous friends during graduate school and talked about my research with them over the course of my graduate career. I would especially like to thank Jacob Robbins, Garret DuCharme and Martijn van Kuppeveld for numerous chats on various theoretical topics that aided me in my path to introduce data science techniques to scanning probe microscopy, as well as graduate solidarity. Talks with Gabriel Caceres have been helpful in aiding my knowledge of time series and machine learning that helped me in my vibration cancellation project. Yuya Ong aided me immensely in helping me understand various aspects of deep learning.

Within my research group, I’d like to thank Lavish Pabbi for his work in inventing ANITA, which I improved by introducing deep learning, as well as taking measurements on the scanning tunneling microscope that I analyzed. Riju Banerjee and Anna Binion also provided STM measurements and collaboration on a number of projects. Kevin Crust was my main collaborator in the topography classification project who did a significant amount of the work, including annotation and hyperparameter tuning. Finally I’d like to thank the DataView team who helped code and plan various aspects of the program. And of course, I’d like to thank my advisor, Eric Hudson, for the immense amount of advice and research support he has offered over the past seven years.
Chapter 1
Introduction

1.1: Motivation

1.1.1: Data Science in Condensed Matter Physics

Data science is a field that uses scientific methods and processes to extract knowledge from data, both in unstructured and structured forms. It has been called the “fourth paradigm” of science, after the first three paradigms of experimental science, theoretical science, and computational science. A hallmark of data science methods, distinguishing them from traditional computational approaches, is that rather than laying out a specific set of instructions for how to achieve some goal, the computer is essentially asked to discover the best approach by making a number of different attempts and being given feedback after each. This discovery process typically requires large quantities of data to test against each method, hence the moniker “big data.” Data driven techniques have matured over the past decade as an integration of statistical and computer science techniques to further scientific progress in a number of fields. Data in disciplines that have lacked solid mathematical theories such as health science and social sciences can now be used to generate powerful predictive models. For example, models generated from computer “read” health records can be used to predict diagnoses. In the social sciences, vocabulary extracted from social media has been correlated with the “big five” personality traits.

Closer to my field of research, advanced theoretical and simulation methods have led to the creation of the materials genome approach to materials discovery. Theory and simulation can provide the ability to decrease the time to find a solution to design and discover new materials. This has led to the development of large, searchable databases to select new material candidates for experimental studies. The introduction of machine learning to the materials discovery improves upon this process. For example, to explore a space of new materials, Meredig et al predicted and ranked the thermodynamic stability of 4,500 tertiary compounds using machine learning in order to discover new, highly stable compounds that hadn’t been investigated yet. Using machine learning in materials discovery is advantageous as it is far less computationally intensive than other methods. Machine learning has also been used for predicting properties and phases of materials, such as the critical temperature of superconductors and metallic glass formation.

In much of the materials-related machine learning research to date, however, including all of the efforts mentioned above, the focus has been on analyzing the output of theoretical work. The vast array of experimental probes of materials systems, and the complex data sets generated by these probes, seems ripe for a machine learning approach. In this thesis I will focus on just one of those techniques – scanning probe microscopy.
1.1.2: Scanning Probe Microscopy

Scanning probe microscopy (SPM) is a branch of microscopy that forms images of surfaces by the use of a probe that scans the specimen. It was founded in 1982, when the scanning tunneling microscope was developed by Binning and Rohrer for which they shared the 1986 Nobel Prize in physics. These techniques are dependent on a feedback loop to control the gap between the probe and sample. The probe is connected to the macroscopic world through various sensors and electronics to record a number of observables. These techniques allow the direct visualization of the structure of matter.

In the last decade, the resolution of these techniques has improved to quantify sub-picometer-level (one trillionth of a meter) displacement of atoms. For example, high-resolution scanning tunneling microscopy (STM) and atomic force microscopy (AFM) provide real-space atomic and electronic structures of material surfaces, visualizing structures of molecular vibration levels, complex electronic phenomena, and chemical bonds. They can provide information on a wide variety of local properties, such as mechanical properties from force-distance curves in atomic force microscopy, and electronic properties from bias spectroscopy in scanning tunneling microscopy.

1.1.3: Data Science in Scanning Probe Microscopy

Unfortunately, the capability to understand and harness experimental information in scanning probe microscopy has thus far been limited. Normally, scientists primarily look for expected phenomena, and accidental discoveries are only made when experimental signals are exceedingly clear. The vast majority of high-quality experimental data is not analyzed, and a smaller fraction yet is published, making these datasets inaccessible to the broader scientific community.

These deficiencies call out for the introduction of advanced computational techniques from data science. Yet only a few groups have taken the plunge in introducing the world of “big data” into scanning probe microscopy. The research group most focused on introducing data science techniques to scanning probe microscopy is Sergei Kalinin’s group at the Institute for Functional Imaging of Materials at Oak Ridge National Laboratory. His group has applied several unsupervised learning techniques – techniques that extract hidden statistical information from multidimensional data – to different types of SPM data. Examples include introducing principal component analysis, spectral unmixing, and the sliding fourier transform, both individually and in combination, such as by combining sliding fourier transforms and spectral unmixing to obtain structural phase information. These techniques have found broad application due to their powerful ability to collapse information in multidimensional datasets into more easily digestible information.

Kalinin has also introduced the idea of “smart data,” the incorporation of machine learning (ML) methods into physical materials research. These methods apply machine learning to analyze image data in order to make predictions, as seen in other fields such as cancer research, and using satellite imaging to predict poverty. In microscopy, machine learning techniques have been used for object recognition in scanning transmission electron microscope (STEM) images, and for extracting chemical information from atomically resolved STEM images. In addition to providing new analysis methods, however, machine learning can also improve the instrumentation and data collection process. For
example, these techniques have been used to enhance the spatial resolution of optical microscopes\textsuperscript{25}. In the context of scanning probe microscopy, machine learning techniques have been used to automatically condition probe tips\textsuperscript{26} and optimize scanning parameters using genetic evolutionary algorithms\textsuperscript{27}.

One may argue that non-ML based techniques exist to perform similar functions. For example, there are a wide array of prescriptive approaches for tuning PID parameters in feedback loops. The techniques referenced above, and what I will discuss in this thesis, are fundamentally different – they are data-centric, and, as mentioned above, allow the computer to essentially discover the best approach. In the end, to determine their usefulness they must be judged relative to these other options, in terms of speed, quality, ease of use, or some other metric.

1.2: Approach

I have approached the problem of introducing novel advanced analytic and data science techniques into scanning probe microscopy in a number of ways. Descriptions of three projects I undertook during my time as a graduate student comprise chapters 3-5 in this dissertation.

The vast majority of time spent during my doctoral studies has been spent solving the issue of viewing and analyzing multidimensional data in the era of big data by leading a multi-institutional effort to create an advanced analytical software package named DataView (I led a local team of programmers creating the core code, while researchers at Harvard, UBC, NIST Gaithersburg and EMPA were responsible for libraries of plug-ins extending core functionality). DataView is an open-source, flexible, user-friendly, multidimensional data analysis software package programmed in Python, specifically designed to handle big data analysis problems in scanning probe microscopy and other applications that involve visualizing and analyzing high-dimensional data. This is necessary as most current SPM analysis software is limited to analysis of 3D datasets (and often only 2D), and is not easily extensible to allow rapid development of new analysis and processing algorithms.

In a second project, I collaborated with an REU student, Kevin Crust, to introduce machine learning on scanning probe microscopy imagery to our lab by building the first steps to aid in the automation of data collection in scanning probe microscopy. Most of the current research done in machine learning on microscopy imagery involves classifying different kinds of objects. This is analogous to classifying an image as being a cat or dog image. We instead investigated the question of whether the computer could perform subjective classification of images similar to what STM experts do continually while taking data – classifying a cat as a “pretty cat” or “ugly cat.” This kind of subjective classification is a crucial first step in the creation of a “self-driving SPM,” and something that would be incredibly difficult to program in a traditional, prescriptive method, as it essentially involves helping the computer develop the experts’ “I know it when I see it” evaluation method.

The third project I’ll describe evolved from a novel, patented algorithm, developed in our lab, to cancel vibrations in the feedback loop of a scanning tunneling microscope.\textsuperscript{28} This is essentially a prediction problem – we predict an internal vibrational signal based on an external accelerometer signal measuring external vibrations. Using my knowledge of time series analysis and machine learning, I improved the
model by applying a nonlinear deep learning model to predict the vibrations, and reduced the mean squared error of the model by an additional 45%.

It should be noted that introducing a new, useful technique doesn’t always bring useful advances. The following are some of the techniques that were attempted but, while we learned about how the techniques could be used, did not yield the scientific advances that we had wished for. I applied unsupervised learning techniques on spectroscopic maps in a number of materials (see Chapter 2 for more details and background) and, while they revealed interesting patterns in the datasets that would otherwise have been hidden, they did not lead to critically important scientific insights for the problems we were studying. However, these are still scientifically useful methods that can be applied to extract additional insight into multidimensional spectroscopies, and I have included these methods in Chapter 2 for further study.

1.3: Structure of Dissertation

This thesis is structured into five separate chapters. The chapters have a similar structure – a motivation section at the beginning, core sections in the middle describing in depth (primarily for future graduate students in the group) different aspects of the projects, and a summary section at the end explaining what we can get from these projects and the future directions of these projects. The first chapter (which you are reading) is the introduction, a high level introduction to the rest of this dissertation, that should, I hope, be accessible to a very broad audience (including my family). This includes the broad motivation in introducing data science techniques to scanning probe microscopy, as well as my approach, and a summary of what the reader is to expect from the rest of the dissertation.

The second chapter deals with the background needed for the rest of the dissertation. The first part of the chapter covers the theory of scanning tunneling microscopy and the types of data extracted from the instrument. The rest of the chapter deals with different aspects of machine learning, the application of which appear in the remained of the dissertation. I introduce the basic concepts and language of machine learning for those who have not been exposed to these concepts. I introduce different methods of unsupervised learning which, while not discussed elsewhere in this dissertation, I did use during my doctoral studies and imagine may be useful as analysis tools for future data analysts in scanning probe microscopy. Finally, I introduce the subfield of machine learning called deep learning, which describes the type of algorithms used in chapters three and four. I end the chapter with a description of what kind of coding is involved in developing a problem-dependent machine learning architecture.

The final three chapters each address one of the projects mentioned above: in chapter three I’ll discuss the extension to ANITA, in chapter four the deep learning classification of image quality, and finally, in chapter five, the development of DataView.
Chapter 2

Background

In this chapter I will first provide an introduction to the theory of Scanning Tunneling Microscopy (STM), the primary instrument used in our research group. I will explain the different kinds of data that can be obtained by STM, and what that data can reveal about the system being studied. Afterwards, I will provide an introduction to Machine Learning, and specifically to two areas within machine learning which have applications to STM experimental data: Unsupervised Learning and Deep Learning. I end the chapter with a description of what kind of coding is involved in developing a problem-dependent machine learning architecture. My aim, aside from giving background information for this thesis, is to bring together this information in one place, easing training for future undergraduate and graduate students in the group.

2.1: Scanning Tunneling Microscopy

The Scanning Tunneling Microscope was the first type of Scanning Probe Microscope created, invented in 1982 by Binnig and Rohrer. It consists of a sharp conducting tip which is scanned over a flat conducting sample. A bias voltage is applied between the conducting tip and surface, such that when the tip is brought within several angstroms of the surface, a measurable (1 pA – 100 nA) current tunnels through the vacuum between them.

2.1.1: Theory of Scanning Tunneling Microscopy

The quantitative theory of the tunneling current in scanning tunneling microscopy is based on Bardeen’s theory. Bardeen’s tunneling theory was published in 1961 and applied to the scanning tunneling microscope by Tersoff and Hamman in 1985. The tunneling current can be calculated using first-order perturbation theory. Assuming the sample is biased by a negative voltage \(( V < 0 )\) with respect to the tip, the fermi level of electrons are raised in the sample, and electrons will flow out of the filled states of the sample to the empty states of the tip. This creates a current \( I = I_{s\rightarrow t} - I_{t\rightarrow s} \), the full form of which is:

\[
I = \frac{4\pi e}{\hbar} \int_{-\infty}^{\infty} |M|^2 \rho_s(E_s) \rho_t(E_t) \{ f(E_s) [1 - f(E_t)] - f(E_t) [1 - f(E_s)] \} d\varepsilon
\]

(2-1)

where \( e \) is the charge on an electron, \( \hbar \) is the reduced Planck’s constant, \( |M| \) is the tunneling matrix element, \( \rho \) is the density of states of either the sample \( s \) or the tip \( t \), and \( f(E) \) is the Fermi distribution:

\[
f(E) = \frac{1}{1 + e^{(E - E_F)/k_B T}}
\]

(2-2)
where $E_F$ is the fermi energy, $k_B$ is Boltzmann's constant, and $T$ is the temperature. The current can be written with respect to the Fermi energy of the sample and tip systems, each set to 0, which are separated by an applied sample bias voltage $V$. These energies become $E_S = \varepsilon$ and $E_T = \varepsilon + eV$. The tunneling current equation (2-1) simplifies to:

$$I = -\frac{4\pi e}{\hbar} \int_{-\infty}^{\infty} |M|^2 \rho_S(\varepsilon) \rho_T(\varepsilon + eV)[f(\varepsilon) - f(\varepsilon + eV)]d\varepsilon$$

(2-3)

This equation can be simplified further as if we are at low temperatures relative to spectral features of interest, as the fermi functions cuts off very sharply at the fermi surface (e.g. at 4.2 K, thermal broadening is of order $k_B T = 0.36$ meV):

$$I \approx -\frac{4\pi e}{\hbar} \int_{-eV}^{0} |M|^2 \rho_S(\varepsilon) \rho_T(\varepsilon + eV)d\varepsilon$$

(2-4)

The tip is typically chosen so that the density of states within the range of fermi energy is flat. When this happens, $\rho_T(\varepsilon)$ can be treated as a constant and taken outside of the integral.

$$I \approx \frac{4\pi e}{\hbar} \rho_T(0) \int_{-eV}^{0} |M|^2 \rho_S(\varepsilon) d\varepsilon$$

(2-5)
Our lab’s scanning tunneling microscopes typically use a PtIr tip, which has a flat density of states in the energy region of interest. Other common tip materials are tungsten (chosen for its hardness) and Pt (chosen for its flat density of states and relative resistance to oxidation).

Bardeen’s tunneling theory is based on several assumptions. It assumes that first order perturbation theory is valid: tunneling is weak enough, and the tip and sample states are orthogonal. Assumptions due to Bardeen’s theory itself include that the occupation probabilities for the tip and sample are independent of each other, and the tip and sample are in electrochemical equilibrium. The matrix element for tunneling is approximately independent of the difference in energy of the two sides of the tunneling barrier. As a result, the matrix element can be treated as constant and taken outside the integral.

\[ I \approx \frac{4\pi e}{\hbar} |M|^2 \rho_t(0) \int_{-eV}^{0} \rho_s(\varepsilon) d\varepsilon \]  
(2-6)

The matrix element \( |M|^2 \) can be described by the fact that both the tip and sample wavefunctions fall off exponentially into the tunneling gap. Approximating the vacuum potential barrier as a square barrier, we can use the WKB approximation to calculate the tunneling probability and obtain \( |M|^2 = e^{-2\gamma} \), with \( \gamma \) given by:

\[ \gamma = \frac{z}{\hbar} \sqrt{2m\varphi} \]  
(2-7)

where \( m \) is the mass of the electron, \( z \) is the width of the barrier, equal to the separation between the tip and the sample, and \( \varphi \) is the potential height of the barrier, a mixture of the work functions of the tip and sample. The STM work function can be measured by recording the tunneling current as a function of tip-sample separation, and this has been done experimentally to see how the work function varies depending on the material of the sample. Most clean materials have work functions of about 4 eV, leading to an exponential dependence of tunneling current on tip-sample separation of about an order of magnitude per Angstrom.

Altogether, the tunneling current can thus be approximated by:

\[ I \approx \frac{4\pi e}{\hbar} e^{-z} \sqrt{\frac{8m\varphi}{\hbar^2}} \rho_t(0) \int_{-eV}^{0} \rho_s(\varepsilon) d\varepsilon \]  
(2-8)

2.1.2: Measurement Types

A Scanning Tunneling Microscope is composed of an atomically sharp tip held a few angstroms above the surface. Fine control of the tip in all three dimensions near the sample is typically controlled by piezoelectric tubes, which can be moved by a user by applying a voltage on them the order of up to 400 V. Figure 2-2 shows the schematic diagram of a scanning tunneling microscope.

We can measure tunneling current as a function of four different variables: \( I(x, y, z, V) \). The variation in tunneling current in \( z \) and \( V \) can be explained by Equation 2-8. The variation in tunneling current in \( x \) and
$y$ can be explained by a position-dependent sample density of states $\rho_s(x, y, \epsilon)$. The STM has a feedback loop mechanism which attempts to hold either $z$ or the current constant at a fixed bias voltage. If we want to know the tunneling current, then we hold $z$ constant. Assuming $z$ is constant, we can then take measurements of the tunneling current as a function of $x$, $y$, and $V$.

### 2.1.2.1: Topography

Topography is the most common type of measurement with an STM. In this mode, the sample has a fixed bias voltage $V_{\text{set}}$ relative to the tip. A feedback loop mechanism is used to hold the tunneling current constant at $I_{\text{set}}$ by controlling the voltage of the $z$ piezoelectric tube. As a result of the strong distance dependence of current discussed above, the STM can effectively map the height of the surface with atomic-scale (sub-pm) resolution.

### 2.1.2.2: Work Function

As seen in Equation 2-8, the current is exponentially proportional to a work function, a convolution of the tip and sample work function. As the material of the tip won’t change, a proper measurement of the STM work function should extract information about the sample work function. By taking measurements of the logarithm of $I$ vs. tip-sample separation and holding the setpoint voltage constant, we can measure the work function by taking the slope of the plot $\frac{d \ln I}{dz}$. In terms of this, the work function can be calculated as $^{36}$:

![Figure 2-2: Schematic Diagram of a Scanning Tunneling Microscope](image) A sharp tip within a few angstroms of an atomically flat surface. Feedback maintains the separation between the tip and sample by holding the tunneling current $I$ constant.
\[ \varphi = \frac{\hbar^2}{8m} \left( \frac{d \ln I}{dz} \right)^2 \]  

(2-9)

This technique is often used to confirm clean vacuum tunneling at the start of an experiment.

2.1.2.3: Differential Conductance Spectroscopy

Equation 2-8 makes clear that if we hold the tip-sample separation constant at a given \((x, y)\) location, and vary the voltage \(V\), the tunneling current is measuring a quantity that is proportional to the integrated density of states. For negative sample bias the electrons are tunneling from the surface of the sample to the tip, and the STM is measuring the integrated density of states below the fermi level of the sample. At a positive bias, electrons are tunneling from the tip to the sample, and the STM is measuring the integrated density of empty states above the fermi surface of the sample.

By measuring the differential conductance of the tunneling current we can instead get a quantity that is proportional to the local density of states of the sample. Taking the derivative of Equation 2-8 with respect to voltage and holding the tip-separation distance constant as \(z = z_0\), we obtain:

\[ g(V) = \frac{dl}{dV} = \frac{4\pi e}{h} e^{-z_0} \left( \frac{\hbar m_0}{\hbar^2} \right)^{\frac{1}{2}} \rho_c(0) \rho_s(eV) \]  

(2-10)

The differential conductance is not typically measured by numerically taking the derivative of the integrated density of states. Instead, we use a lock-in amplifier to modulate the bias voltage around a voltage of interest, and then measure the resulting current modulation, which is proportional to the differential conductance. This can be seen by applying the Taylor expansion to the current:

\[ I(V + dV \sin \omega t) \approx I(V) + \frac{dI}{dV} \bigg|_V \cdot dV \sin \omega t \]  

(2-11)

Thus, at a given point on the surface, a differential conductance spectrum obtained by measuring the amplitude of the lock-in output is proportional to \(dI/dV\), and in turn, is proportional to the density of states of the material as a function of sample bias (energy). This form of measurement is commonly called Scanning Tunneling Spectroscopy (STS).

2.1.2.4: Differential Conductance “Spectral Survey”

The density of states can vary with position, not just with energy. We can measure how the differential conductance varies with position by measuring it as a series of points on a surface. At a given spatial position, the differential conductance is measured at multiple energy points. Afterwards, the tip is moved to a different position using the feedback as in topography mode to the next position. Then, the feedback is disabled and the next series of conductance measurements are made. This creates a
“spectral survey” \( g(x, y, V) \), a three-dimensional dataset. This survey technique allows us to visualize inhomogeneities in the density of states of the surface of the material.

Typically, when taking a differential conductance spectrum, either individually or in a map, this measurement is repeated a number of times. In addition, the voltage is ramped in two directions: from a maximum positive voltage to a minimum negative voltage and back again. This allows better characterization of the uncertainty in the measurement. Altogether, our conductance dataset will typically have up to five dimensions: \( g(r, d, x, y, V) \) where \( r \) is the “repeat number” and \( d \) is the direction of the scan. A summary of the types of datasets and their typical sizes is provided in appendix B.

### 2.2: Introduction to Machine Learning

Machine learning (ML) is a field of computer science that uses statistical techniques to give computers the ability to “learn” from data, without being explicitly programmed.\(^{38}\) It is a subset of artificial intelligence. The name *machine learning* was coined in 1959 by Arthur Samuel.

Below I provide a basic introduction to the major concepts in ML. For more in-depth information, I recommend *An Introduction to Statistical Learning* by Gareth James et al.\(^{39}\) as an introductory resource, and *The Elements of Statistical Learning* by Trevor Hastie et al.\(^ {20}\) as a reference resource.

Mitchell\(^ {40,41}\) provides the definition of *learning* in machine learning as follows: “A computer program is said to learn from experience \( E \) with respect to the class of tasks \( T \) and performance metric \( P \), if its performance at tasks in \( T \), as measured by \( P \), improves with experience \( E \).” This encompasses a broad amount of experiences, tasks, and performance measures.

In classical programming, a human inputs rules (a program) and data to be processed according to the rules, and from the output of the program comes answers. Using machine learning, a human inputs data as well as the answers expected from the data, and from the output of the program comes rules (See Figure 2-3). These rules can then be applied to new data to produce original answers.

Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. Machine learning tasks are usually described in terms of how the machine learning system should process a *sample*. A sample is a collection of *features* that have been quantitatively measured from some object or event that we want the machine learning system to

![Figure 2-3: Machine Learning: A new programming paradigm\(^ {37}\).](image)
process. Typically, the sample is represented as a vector $\mathbf{x} \in \mathbb{R}^n$ where each entry $x_i$ of the vector is a feature.

### 2.2.1: Tasks

A task $T$ is usually described in terms of how the machine learning algorithm processes a sample. There are many different kinds of tasks that can be solved with machine learning, but the most common tasks are classification and regression.

Classification is a task in which a computer program specifies which of $k$ categories an input belongs to. An example of classification is object recognition, in which the input is an image, and the output is a numeric code which identifies the object in the image. The learning algorithm thus needs to produce a function taking an input vector (such as an image of a cat) to an output, either a single numeric value indicating the most likely category (e.g. 1 which we have associated with “cat”) or a vector (e.g. $[\text{dog-probability}, \text{cat-probability}] = [0.1, 0.9]$), indicating the probability distribution over the categories. The simplest classification algorithm is logistic regression, often seen outside of machine learning in the context of statistical inference\(^4^2\). As an example of its use with a single variable, imagine binarizing data – that is, taking a continuous variable, like the brightness of a greyscale pixel in an image, and deciding whether to make it black or white. If you knew what some pixels should do, you could fit to those, allowing you to or offer a likely value, or the probability for each value, for the remaining pixels.

There are many potential examples of the use of classification in STM. For example, from a topography and spectral survey, one could potentially classify the local or global phase of a material, or identify the locations and identities of atoms, for example dopants or impurities\(^2^4\). In this thesis (Ch. 4) I’ll describe an application of classification to STM topographies, identifying whether they have atomic resolution or not, and what their quality is.

Regression is task in which a computer program predicts a numerical value given a sample input. This is similar to classification, but the format of the output is different, as the output is a continuous rather than discrete variable (or vector thereof). A real world example of a regression task is the prediction of housing prices from properties of the houses, as seen in the Ames housing dataset\(^4^3\). Linear regression, also used widely outside of machine learning\(^4^4\), is a simple example. Potential examples from STM include extracting information like gap size from superconducting spectra or local wavelength from inhomogeneous charge density wave materials.

### 2.2.2: Performance Metrics

Evaluating the abilities of a machine learning algorithm requires a quantitative measure of its performance. This performance metric $P$ is specific to the task $T$ being carried out by the system. Typically, the performance measure is evaluated on a test set, a dataset that is independent from the training set, the dataset which the model is using for learning. This section will describe different performance measures typically seen in classification and regression tasks. For a regression task, the following metrics are commonly used:
• **Mean Squared Error**: The averaged squared distance between the predicted values \(y_i\) and the true values \(\hat{y}_i\):

\[
MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2
\]

(2-12)

Mean squared errors heavily weigh outliers, as a result of the squaring of each term, which weights large errors more heavily than small ones. This property has led researchers to use the next performance metric.

• **Mean Absolute Error**: The average absolute distance between the predicted values and the true values:

\[
MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|
\]

(2-13)

For a multilabel classification problem, a way to visualize the performance of a model is a confusion matrix. Figure 2-4 shows a visualization of a confusion matrix, along with the three main metrics for classification based on this matrix. Each item is placed in the matrix based on its true category (column) and its predicted category (row). Note that the number of items actually in each category can vary (hence the different widths of the two table columns) as can the number of items predicted to be in a given category (hence the different heights). The following metrics are commonly used, each defined as a ratio of a “good” subset (green) over a “total” subset (green + red), either for the whole dataset, or for each category individually:

• **Accuracy**: Accuracy is the fraction of samples for which the model produces the correct output. For a multilabel classification problem, accuracy can either be subset accuracy (the set of labels predicted for which a sample exactly matches the corresponding true labels) or average accuracy (the average accuracy per label)

• **Precision**: Also known as positive predictive value. Precision is the proportion of true positives (items that are predicted to be in a category and are actually are) divided by the total number of predicted positives:

\[
\text{precision} = \frac{TP}{TP + FP}
\]

(2-14)

where \(TP\) is the number of true positives, and \(FP\) is the number of false positives, which are those that are predicted to be in a category but aren’t. Accuracy may not be a reasonable metric for classification if the labels in the dataset are unbalanced. High precision means that a model has returned substantially more relevant results than irrelevant ones. Note that unlike accuracy, which is a statement about the total categorization, each category gets its own precision score.
Recall: Also known as sensitivity. Recall is the proportion of true positives divided by the total number of actual positives:

\[
\text{recall} = \frac{TP}{TP + FN}
\]  
(2-15)

where \( FN \) is the number of false negatives, which are those that are not predicted to have the label but actually have the label. High recall means that a model has returned most of the relevant results.

F1 Score: A metric that considers both the precision and recall together into a single metric:

\[
F_1 = \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]  
(2-16)

This is useful when one wants to optimize the precision and recall of a classification task simultaneously.

In addition to performance measures, which let us know how well the algorithm has achieved its task, we also must define a loss function for each task, which may be distinct from the performance measures. Loss functions are used to optimize a machine learning algorithm. Linear regression, the simplest form of regression, minimizes the mean squared error of the training dataset, but there are a number of other forms of regression which use different loss functions. An example is Ridge Regression, a form of regression which adds an \( L^2 \) norm of the feature matrix to the mean squared error\(^47\). However, Ridge Regression’s performance is still measured using the mean squared error on the test dataset. A clearer picture of the use of loss functions will emerge in the detailed discussions of the learning process below.
2.2.3: Generalization

Machine learning is data-driven, and the data used to build the final model typically comes from multiple datasets. A model ought to generalize well, or perform well on data it has not seen before. In order to achieve this, not all data is used for training, but some is “held out” – used after training to determine the success of the training. In practice this held out data is itself used for two different purposes, hence the definition of three different types of datasets used to build a model: the training dataset, the validation dataset, and the test dataset.

The training dataset is the dataset that the machine learning algorithm learns from. It is used to fit the parameters of the model by minimizing the error of the model with respect to this dataset. For a traditional linear regression, for example, this would be all of the data, and the quality of fit would be calculated using the same data from which the model was determined.

But in ML, with its ability to make very complex fits (with a large number of free parameters), there is a danger of overfitting the data (Fig. 2.5) Testing after training with held out data allows us to identify when this is happening. When overfitting occurs, the error when applying the model to the training data will become very low, while the error applying the model to held out data will remain (relatively) high. Looking for this behavior is important when creating and training an ML model.

As mentioned above, we divide the held-out data into two datasets – the validation dataset and the test dataset. This reflects the methodology of creating a ML model. It will not immediately be obvious what architecture will work best for a given set of data. To determine this, a number of different models are created using different “hyperparameters” (the details of these parameters are discussed later). A similar process in the language of non-ML fitting would be choosing which curve to fit to the data (what order polynomial, or how many gaussians, ...). Each of these models are then tested against a validation dataset, in order to give an unbiased view (not using the training data) of which of the various models works best. More than that, however, the results of testing against the validation dataset helps drive the
search for the best hyperparameters. In this way, the validation dataset in a sense becomes a training dataset – training the architecture of the system as a whole, rather than the details of a particular model. Again, to relate back to conventional fitting, for a polynomial fit \( y_i = \sum_{k=0}^{n} c_{n,k} x_i^k \), the details of the model \((c_{n,k})\) is directed by the training data, while the hyperparameter \((n, the order of the fit)\) is directed by the validation dataset. Thus, in the end we also need a final test dataset. Just as the validation dataset can tell us if a given model is overfitting, the test dataset ensures that the training and validation datasets aren’t driving the architecture to an ungeneralizable result.

The proper choice of validation and test datasets is important. At least they should follow the same probability distribution as the training dataset. But a number of methods can be used in making this choice. The easiest way is holding out data, where part of the original data is set aside for later testing. A more sophisticated method is cross-validation, where the holding out process is repeated by creating a number of partitions of original data, using some for training and the rest for testing. A common version of this is K-fold cross validation. Here the original dataset is divided into \( k \) parts, with one part used for testing and \( k-1 \) parts used for training. This is repeated \( k \) times to obtain a distribution of the performance metrics, and tends to make a better check on generalization than holding out a single testing set.

### 2.3: Unsupervised Learning in Scanning Probe Microscopy

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning progress. Unsupervised learning, or “learning without a teacher”, involves directly inferring the statistical properties of a dataset without the help of a supervisor or teacher providing correct answers for each observation\(^{20}\). Unsupervised learning deals with situations where we only have a set of features \( X_1, X_2, \ldots, X_p \) measured in \( n \) observations. There is no interest in prediction, because there isn’t an associated target variable. The goal is to discover interesting things about the measurements. Example goals include an informative way to visualize the data, or discovering subgroups among the features or among the observations. Unsupervised learning refers to a diverse set of techniques for answering questions such as these, some of which I’ll exemplify below.

Applied to STM data, and in particular to differential conductance spectra, unsupervised learning techniques allow us to unravel statistical patterns which are not immediately obvious to the human eye when observing datasets via traditional methods (either looking at a series of spectra sequentially, or looking at the spatial dependence of constant energy maps). As the data in spectroscopy is typically unlabeled at first, these methods are a useful way to perform exploratory data analysis to search for hidden structures within the data.

#### 2.3.1: Principal Component Analysis

Principal Component Analysis (PCA) is a statistical procedure which uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly
uncorrelated variables called *principal components*. PCA serves as a tool for data visualization and data compression\(^9\).

PCA finds a low-dimensional representation of a dataset that contains as much of the variation as possible. While each of the \(n\) observations lives in a \(p\)-dimensional space, not all of these dimensions are equally interesting. PCA seeks a small number of dimensions that are as interesting as possible, where the concept of *interesting* is measured by the amount that the observations vary along each of the \(p\) features. The first principal component *score* is defined as the normalized linear combination of features that have the largest variance:

\[
t_{i1} = w_{11}x_{i1} + w_{21}x_{i2} + \cdots + w_{p1}x_{ip}
\]  (2-17)

\(i\) refers to the sample values of each feature. Normalized here means that \(\sum_{j=1}^{p} w_{j1}^2 = 1\). These elements \(w_{11}, \ldots, w_{p1}\) are referred to as the *loadings* of the first principal component. Together, the loadings make up the principal component loading vector: \(\mathbf{w}_1 = (w_{11}, w_{21}, \ldots, w_{p1})^T\). The geometric interpretation is that the loading vector defines a direction in feature space along which the data varies the most, as seen in Figure 2-6, and that \(t_{i1}\) is the projection of the \(i\)th sample onto \(\mathbf{w}_1\).

After the first principal component \(\mathbf{w}_1\) has been determined, we can find the second principal component \(\mathbf{w}_2\). The second principal component is the linear combination of \(x_1, \ldots, x_p\) that are *uncorrelated* with \(\mathbf{w}_1\). The second principal component scores take the form:

\[
t_{i2} = w_{12}x_{i1} + w_{22}x_{i2} + \cdots + w_{p2}x_{ip}
\]  (2-18)

Constraining \(t_2\) to be uncorrelated with \(t_1\) is equivalent to constraining the loading vector \(\mathbf{w}_2\) to be orthogonal to \(\mathbf{w}_1\). Further components repeat this process. Thus, PCA can be thought of as a “rotation” of the dimensions of a dataset in such a way that the earliest components explain most of the variability of the data.

---

**Figure 2-6**: PCA of a dataset with a multivariate Gaussian distribution\(^8\). The vectors shown are the principal component loading vectors, centered on the mean of the data. The first loading component is in the direction of maximum variance of the distribution, while the second loading component is orthogonal to the first.
PCA enables easier visualization of otherwise unwieldy datasets by plotting the principal component scores of all the samples in a scatterplot in a low dimensional setting, as the first principle components should hold most of the information of the dataset. If a dataset is labelled by class, it is possible to view clusters of data in the PCA plot.

To visualize how much variance the principal components are holding, a researcher can create a scree plot (Fig. 2-7(c)), a line segment plot showing the fraction of total variance explained by each principal component.

In the context of scanning probe spectroscopy, a spectroscopic survey of \(N \times M\) pixels, each with \(P\) points (e.g. energies) is represented in PCA as a superposition of the eigenvectors \(w\) (our new orthogonal spanning set):

\[
g_{ij} = \sum_{k=1}^{P} t_{ik}w_{jk} \tag{2-19}
\]

Here \(i\) refers to the sample, so is equivalent to \((x,y)\), \(j\) refers to the energy so is equivalent to \((E)\), and \(k\) refers to the component order. Applying PCA on STS thus returns us a two-dimensional map of each of the scores \(t_k(x,y)\) of the principal components, and the eigenvectors \(w_k(E)\) (which can be interpreted as spectra as a function of energy). The first eigenvector contains the “most” spectral information, and the first \(p\) scores \(t_k(x,y)\) contain the majority of information within the dataset, while the remaining \(P - p\) sets are dominated by noise. It is important to note that PCA describes statistical information of the spectroscopy, rather than physical information, so a researcher has to be careful in interpreting results from PCA. Figure 2-7 shows an example of visual representations of PCA data from a real STS.

---

**Figure 2-7**: PCA visualizations of a BSCCO DOS map with 81 energy points. (a) The first principal component scores, plotted spatially, contains the majority of the information of the DOS map. (b) First three PC eigenvectors \(w\). (c) Scree plot, showing the cumulative explained variance vs. number of principal components. (d) 2D visualization of the dataset, plotted by the first two principal component scores. The colors show how one might go about using the PCs to segregate clusters in the data (though this data is not well segregated by PCA, an analysis of the clusters pictured here is presented in Fig. 2-10 as a demonstration of usage of the technique).
dataset, a spectral survey of a BSCCO sample. The principal components $t_k(x,y)$ retain spatial correlations of the full three-dimensional dataset. PCA can be very helpful in identifying different “species” within a large set of data. This often becomes clear by making 2D plots of the first two principle components of every sample (as in Fig. 2-7d). PCA will often separate data into different clusters (unfortunately, for the pictured data this doesn’t happen – the different colors show how you might try to separate the data based on their first two principal component scores $t_1$ and $t_2$).

2.3.2: Spectral Unmixing

Whereas PCA is essentially a rotation of the data from the original coordinate system (e.g. versus energy) into a new coordinate system (versus principle component number), where all information about the data is preserved, spectral unmixing instead assumes that there are a relatively small number of “endmembers” (e.g. principle components) that actually describe the data, and that any part of the data not described by a mixture of those endmembers is noise. It was developed for the analysis of hyperspectral images – very similar to spectral surveys of STM, except with frequency (color) in addition to position instead of energy. In the context of geospatial imaging, hyperspectral data is often used to determine what materials are present in a scene, such as roadways, vegetation, or water. Each pixel can be interpreted as a mixture of spectra of several materials. Spectral unmixing refers to the process of unmixing one of these ‘mixed’ pixels into a number of material spectra, called endmembers, and the proportion of each endmember in every pixel, called abundances $^49$.

Although there are many different methods of doing spectral unmixing, here I’ll focus on Bayesian linear unmixing. Although it is slow, and additional insight is needed to optimize the algorithm, an algorithm by Dobiegeon et. al. $^50$ has been used to analyze SPM data$^{17,51}$, so it is worth mentioning here. The Bayesian approach assumes data in a form $Y = MA + N$, where the observations $Y$ are a linear combination of endmembers $M$ weighted with relative abundances $A$, corrupted by Gaussian noise $N$. Like all Bayesian linear unmixing algorithms, the researcher must at least fix the number of endmembers. If this is unknown, a reasonable starting point would be to use PCA to identify how many spectra are needed to capture a majority of the spectral variation. The algorithm starts by initially projecting endmembers using the N-FINDR algorithm$^{52}$, with an initial guess of the abundances using multiple least squares regression. Using this initial guess is a faster estimation for linear spectral unmixing than using the full Bayesian algorithm, but the N-FINDR algorithm assumes that each endmember has at least once pixel that is “pure”, or have an abundance of 1, which may not be the case in the dataset. The full Bayesian algorithm estimates the endmember spectra and abundances jointly. The Dobiegeon algorithm further assumes that endmembers and abundance coefficients are non-negative, additive, and sum to one$^{53-55}$.

In the context of STM, it must be emphasized that the construction of a spectrum from endmembers is additive in linear unmixing (just as in PCA), thus this algorithm only makes sense in contexts where the spectroscopy can be thought of as being additive of different components. Linear unmixing has been applied to scanning tunneling spectroscopy of iron-based superconductors$^{14}$ and topography, after converting a topography $z(x,y)$ to a map of structural spectroscopies $z(x, y, k_x, k_y)$ using a shifting fourier transform and applying linear unmixing on the wavevector axes$^{19}$ to extract multiple structural lattices present in a system. Figure 2-8 shows an example of spectral unmixing (using just the N-FINDR
discovered endmembers and multiple least squares determined abundances, as discussed above) on a BSCCO dataset.

Figure 2-8: Linear Unmixing of a BSCCO DOS map into three endmembers. Same dataset as Figures 2-4 and 2-5. (a) Endmember spectra $m_i(E)$; each pixel is interpreted as a linear superposition of these spectra. (b-d) Abundance maps $a_i(x, y)$ for each of the endmembers in (a). These could potentially be interpreted different electronic states in the system.

2.3.3: Clustering

Clustering refers to a series of techniques for finding subgroups in a dataset called clusters. In clustering, we seek to partition observations of a dataset into distinct groups so that the observations within each group are quite similar to each other, while observations in different groups are quite different from each other. Clustering requires that we define what it means for two or more observations to be similar or different from one another.

Figure 2-9: Results of clustering of a dataset. Squares are separated into three clusters.
K-means clustering is a specific clustering algorithm for partitioning a dataset into $K$ distinct, non-overlapping clusters. The algorithm requires us to input the number of clusters $K$ (which, as in choosing the number of endmembers for spectral unmixing, is rarely obvious, and will be discussed below). It partitions the observations into clusters in which each observation belongs to the cluster with the nearest mean.

K-means clustering is said to be good clustering when the within-clustering variation is as small as possible. This requires defining a distance. For example, if the Euclidian distance is used (a common but by no means only choice), the within-cluster sum of squares would need to be minimized:

$$\operatorname*{arg\,min}_{i=1}^{k} \sum_{x_j \in S_i} \| x_j - \mu_i \|^2$$

(2-19)

Where the $\mu_i$ are the mean of the points $x_j$ in each cluster $S_i$.\(^{57,58}\)

For analyzing scanning tunneling spectroscopy data, this distance metric isn’t ideal. Absolute STS conductance values are, in a sense, arbitrary. This is because when setting the STM junction voltage and current, the conductance spectra between zero and the voltage setpoint is guaranteed (within noise) to integrate to the current setpoint. If the setpoints are modified, the magnitudes of the spectra will change – we will change the proportionality between the conductance and the density of states. To first order, however, the shape of the spectrum should remain the same. Thus in nearly all cases we treat conductances with the same shape but different overall amplitude as identical. To implement this notion we normalize the conductances by dividing by the $L^2$ norm (Euclidian length squared: $\sum g_j^2$) of the conductance. This results in a distance metric called the “cosine distance” (where the similarity between two vectors depends on their relative angle in the vector space, rather than on their magnitudes).

As the number of clusters in K-means clustering may not be known in advance, a metric should be used to select the number of clusters. A point in a cluster should ideally be similar to other points in the same cluster and different from points in other clusters. Silhouette analysis is a way to visualize this and study the separation distance between the resulting clusters\(^{59}\). For each sample, the average distance is computed between it and other points in its assigned cluster and compared to its average distance to points in the next closest cluster, yielding a “silhouette coefficient” between -1 and 1, where 1 indicates the sample is much closer to other samples in its assigned cluster and 0 means it is equidistant, on average, to points in two different clusters. Negative scores indicate the sample may have been assigned to the wrong cluster. The average score across the whole dataset is a measure of how well the clustering is done as a whole. Figure 2-10(b) shows use of this technique on clustering of the BSSCO data shown in the earlier Figures 2-7 and 2-8.
So far we’ve discussed unsupervised learning techniques, where the computer is tasked with seeking patterns in the data without (as much as possible) human intervention. Supervised learning algorithms, on the other hand, are given, in addition to the dataset features (i.e. the conductance at a number of energies) a label, or “target” provided by a human teacher. The computer’s job in supervised learning then is to figure out how to get from the features to the target. Supervised learning is the most common task in machine learning, as being able to predict targets from interesting features, such as the price of a house from different aspects of a house, is very useful.

A central problem in machine learning is the ability to meaningfully transform data – that is, to learn useful representations of the input data at hand that get us closer to the expected output. A representation is a view of one subset of information about the data. For example, in studying pictures and trying to figure out their content, it may be useful to find all the edges in the picture, or separate the high contrast and low contrast regions of the image, or so forth. Maps of each of these would be representations. In the early days of ML, a machine learning engineer had to manually transform features of data to useful representations in order to make better predictive models.

Deep learning is a specific subfield of machine learning which allows us to do this automatically. It allows the computer to itself learn how to create useful representations from the data. Though the details of how this is done is beyond the scope of this thesis, one important aspect is that the representations come in successive (hopefully increasingly meaningful) layers, with each layer containing representations built upon the representations in the previous layer\(^37\). So, for example, following our image identification example from above, we may have an “edges” representation of the original data in the first layer, and a “contrast” representation of this “edges” representation in the second layer (both among many others). The “deep” of deep learning is a reference to this idea of successive layers of

---

**Figure 2-10**: Cluster visualization of the same BSCCO DOS map as Figure 2-6. (a) Clusters of the data created with K-means clustering with 3 clusters. DOS map is $L^2$ normalized before clustering. (b) Silhouette plot for the clusters in (a), showing how well each cluster is identified. The average score (red dashed line) is a measure of the overall quality of the cluster (closer to 1 is better; averages under 0.5, as here, typically indicate that the clusters are artificial - not indicative of real clustering).
representations. Modern deep learning models can involve dozens of layers of representations that are learned automatically from exposure to training data.

Deep learning models are inspired by information processing and communication networks in biological nervous systems, and are also called artificial neural networks. However, they have various differences from the structural and functional properties of biological brains.

Figure 2-9 shows the framework of deep learning when used in supervised learning. Input data is fed into the model, which feed-forward into later layers. Each layer has a number of parameters called weights which characterize and transform the features of the previous layer. A layer transforms the data from the previous layer by applying a non-linear activation function on top of a linear transformation of the previous data. After the data has gone through all the layers, it makes a prediction \( \hat{y} \), which is compared to a true label \( y \). The predictions and true labels are computed in a loss function to create a loss value. This loss value is used by an optimizer to update the weights of the deep learning model using a process called backpropagation such that the loss function is minimized.

Backpropagation is a method used to calculate the gradient needed to update the weights of a neural network, and is essentially a form of chain rule to iteratively obtain gradients for each layer. Gradients are needed because the optimizer typically uses a form of gradient descent, an algorithm which finds the minimum of a function.

2.4.1: Dense Neural Networks

A Dense Neural Networks (DNN) is the most straightforward type of neural network, but also the one with the most parameters. A layer in a dense neural network is composed of a number of units \( x_j \), each of which are connected to all of the units of the next layer \( y_i \) with some weight \( w_{ij} \). Thus, to compute the value of a unit in a layer, a weighted sum over all of the units in the previous layer is first computed, a bias (constant \( b_i \)) is added, and an activation function \( \varphi \) is applied:
An activation function, typically nonlinear, is used to introduce extra complexity into the model. Common examples of activation functions include the logistic (or sigmoid or s) function, which maps the output into the range $-1$ to $1$, and the rectified linear unit (ReLU) function, which is defined as zero for negative inputs, and the input for non-negative inputs. ReLU activation functions, modelled loosely on the behavior of actual neurons (off, then slowly turning on) are widely used in continuous output tasks while the logistic activation function, which constrains the output to a fixed range, is often used at the end of classification problems.

Because of the large numbers of connections, dense neural networks are typically used with vector data, where each sample is encoded as a vector of features. They have the disadvantage of having a large number of parameters, so they can easily succumb to overfitting. Thus they are much less often used in data with spatial dependences, due to the likelihood of swelling numbers of connections. However, due to their ease of use, dense models are commonly used at the output stage of other types of neural networks (as, for example, in the connections between the “Hidden” and “Output” in Figure 4.12). Their simplicity also allows relatively easy investigation of the final model (by viewing each layer of connection weights as an image).

2.4.2: Convolutional Neural Networks

As opposed to DNNs, Convolutional Neural Networks (CNNs) are a type of deep learning network used to process spatial data, such as images. They are inspired by the visual cortex of animals. In the visual
Cortex, neurons respond only to stimuli in a restricted region of the visual field known as the receptive field. Receptive fields of different neurons partially overlap so that they cover the entire visual field.

CNNs are composed of a number of different types of layers. The first and most important type of layer in a CNN is a **Convolutional Layer**. These layers apply a convolution operation to their input, creating feature maps. This layer emulates the response of individual neurons to visual stimuli. Convolutional layers learn local patterns of their input feature space, unlike dense layers, which learn global patterns. A convolutional layer can learn translationally invariant patterns, so that once a certain pattern has been found in an image, the layer can find it elsewhere in an image as well. They can also learn spatial hierarchies of patterns. For example, later convolutional layers can build upon earlier features like edges to create body parts such as eyes or ears in the process of learning how to detect an animal from an image. Convolutional layers can be one- or multi-dimensional depending on the nature of the dataset. The weights in a convolutional layer determine the convolutional filters.

The second type of layer in a CNN is a **Pooling Layer**. This type of layer downsamples feature maps from a convolutional layer. Typically the type of operation used for pooling is max-pooling – it takes a grid of points (like a 2 x 2 grid) and returns as a value of the maximum value of these points when downsampling the feature maps. Pooling is used to reduce the number of parameters of the model, which helps reducing overfitting. It also allows the development of a spatial hierarchy of features by successively applying a single kernel to increasingly larger fields of view in the image – subsampling the image so that the same number of pixels refers to a larger physical region (Fig. 2-13).

CNNs also typically use dense, fully-connected layers (section 2.4.1) at the end of the network. The output of the model needs to be a dense layer with the proper activation function - no activation function for regression, and something like a logistic function for a classification.

Convolutional Neural Networks have been incredibly successful in solving previously unapproachable problems in computer vision. They have been used to help identify cancer and predict poverty from satellite imagery. Reverse image searches, in which an image may be uploaded into a search engine, like google, and other similar images identified, have been enabled by CNNs. In the context of microscopy imagery, they have been used to detect defects in scanning transmission electron microscopy (STEM) images, to improve the resolution of optical microscopy, and to detect common

---

Figure 2-13: Typical Convolutional Neural Network Architecture. A convolutional layer convolves the image with a variety of kernels to create series of feature maps. A pooling layer then subsamples these feature maps (reduces the number of pixels in them) so that later convolutional layers with the same kernels can look at larger windows of the data. For example, a kernel the size of the red box in the original image will cover twice the physical area of the image after subsampling (red box at right).
features in scanning electron microscopy (SEM) images. I will discuss our use of a 2D CNN on STM topography data to predict atomic resolution and image quality in Chapter 4.

2.4.3: Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of deep learning network designed to work with sequences of data. Traditional neural networks can’t remember information about previous data. Imagine a researcher might want to classify an event that is happening at each point in time – information about previous events might be important to predict later events, and traditional neural networks can’t handle this. Recurrent neural networks are the solution – they are networks with loops in them, allowing information to be passed from one step of the network to the next. Figure 2-14 shows the basic structure of an RNN, as well as what an RNN looks like when it is unraveled, as an RNN can be thought of as multiple copies of the same loop. The “state” of the system, as determined by the previous inputs and outputs, persists in the RNN unit and is passed to later iterations of the unit as new input data is passed to it. The details of what a “state” is depends on the particular implementation of the RNN (in general the new state is calculated with some activation function, based on the current state and the input).

The chain-like nature of RNNs shows that they are intimately related to sequences and lists, and are the natural architecture of neural network to use for these types of data. Recurrent neural networks have been successfully applied to many different problems, including speech recognition, language modeling, translation, and image captioning.

There are a variety of different implementations of RNNs. The original, or “Vanilla” RNNs, are essentially single layer neural networks, where at each time step the input and current state form the input and the next state and output form the output. Because of the use of nonlinear activation functions, it turns out these don’t do particularly well capturing long term dependencies (the state tends to be much more responsive to new information than to old information). Thus, other types of RNNs have been implemented.

The Long Short Term Memory (LSTM) network invented by Hochreiter et al. is a complex solution to the problem of poor long term memory. LSTMs have a large internal structure compared to vanilla

![Diagram of RNN structure](image)

Figure 2-14: Structure of a Recurrent Neural Network. RNNs are networks with loops in them, allowing information to persist. A chunk of neural network A looks at some input value $x_t$ and outputs a value $h_t$. This system state persists in A, being fed into the next node along with new data. An RNN unit can be thought of as multiple copies of the same unit, each unit passing information to the next.
RNNs, composed of layers which interact with each other in ways so that long term dependencies can be remembered. In effect, they replace the single layer neural network of the Vanilla RNN with something more akin to the convolutional layers of a CNN, enabling more complex handling of memory (instead of a single state, the system ends up with multiple states, deciding how much of each to forget and how much to pass on).

However, because of their large internal structure size, LSTMs typically take a long time to train and require large amounts of data (compared, for example, to Vanilla RNNs). A Gated Recurrent Unit (GRU) is a variant on the LSTM which simplifies its structure (reducing, for example, the decisions about keeping and forgetting information about previous states)\textsuperscript{73}. With the simpler structure and fewer parameters comes a faster training time, and better performance than LSTMs, especially on smaller datasets\textsuperscript{74}.

In the context of STM, recurrent neural networks are useful for analyzing and predicting time series of different signals, including, sequences of images (combining a CNN for 2D image analysis with an RNN for the evolution analysis of the images, for example, could allow the computer to analyze the progression of image improvement for better tuning of scan parameters). In Chapter 3 I will examine the use of an RNN on an external mechanical vibration signal to predict vibrational data in the Z-feedback signal of an STM.

### 2.4.4: Deep Learning Process

In the final section of this background chapter I will explain in more detail the process required to effectively use deep learning techniques. The basics of the ML techniques themselves (and references) can be found in the earlier introductions in this chapter; they are omitted here in the interest of brevity. Deep Learning in practice takes more than just feeding data into a black box to return answers. It involves a number of steps, from preprocessing data, to model creation, to model optimization, always with an eye toward generalization on new examples. Figure 2-15 shows the workflow of how a scientist crafts a deep learning model.

In general, by using a machine learning process the researcher makes two main hypotheses about their dataset: the outputs can be predicted given inputs, and the data available is sufficiently informative to learn the relationship between the inputs and outputs. This is similar in concept to deciding to attempt a linear fit of a dataset, and just as with that decision, the researcher will receive feedback during and
after the learning process to indicate whether these hypotheses are accurate.

The first step in deep learning is defining the problem and assembling a dataset. The researcher must define the input data and what they are trying to predict. The researcher must also decide what type of problem they are facing – whether it is binary classification, multiclass classification, regression, or other task.

The second step is choosing a measure of success. To control something, you need to be able to observe it, and you must define what we mean by success. This means determining the type of performance metric for the problem at hand, and choosing a specific loss function, which is what the model will optimize.

The third step is deciding on an evaluation protocol, in which the researcher must establish how they’ll measure their current progress. Common evaluation protocols include maintaining a hold-out validation set, typically done when you have plenty of data, and K-fold cross-validation, used when you have too few samples for hold-out validation to be reliable.

At this point we have begun to craft the model. We should know from the first step what class of neural network will be used (CNN for feature analysis in images, RNN for time series analysis, and so forth), which is enough to move to the fourth step - data preprocessing. How to prepare the data depends on both the model to be used and on the format of the data itself, for example, whether the data is numerical vector data, image data, or sequential data. Typically, data has to be modified before being fed into a machine learning algorithm. An example of preprocessing is feature scaling, in which the features (or independent variables) of the data are modified by subtracting the mean of the feature and dividing by the standard deviation. This is often done to ease the learning process, preventing Euclidian distance calculations from being dominated initially by a handful of features simply because of their scaling.

In thinking about training sets, data augmentation is also often used to both enlarge the training set and generalize the input. For example, for an image object classification task (what is the salient object in this image?), while starting with a large number of tagged images (images where the answer is known) is useful, this dataset can be augmented by doing transformations such as randomly rotating, translating, or flipping the images. Not only does this increase the size of the training set, it also helps achieve our goal of having the computer recognize objects which are tilted, off-center, or facing a different direction, so preprocessing the data in this fashion will help create a more generalizable model.

Now we get to actually building up, using and testing the model. Although there are general concepts in this step regardless of the type of model being used, the details do vary, so for ease of discussion I’ll focus on a binary classification problem (is the input data “true” or “false”?) The goal at this stage is to develop a small model that is capable of beating baseline (random guessing). For example, if 50% of the data is tagged true, then the model ought to perform better than 50%. Assuming the initial hypotheses about the data were true (that they are predictive and sufficient for learning), we should be able to do this.

At this point we are beginning to nail down the architecture of the model. That is, in addition to the type of neural network, the loss function and validation protocol determined earlier, we need to define the details of the model. For example, what activation functions will be used and how many layers will we
have? The choice of these parameters is in a sense art, driven by some knowledge of the data and experience in using the models. This is similar, for example, to choosing a fitting function for complex data – you may have an idea of how many Gaussians will need to be added to describe the data, but you’ll know better once you’ve played with it a little and get feedback from the process. And, to be clear, though the power of deep learning is that the computer can make many decisions on its own about how best to extract information from the data, the overall architecture still must be developed by an experienced practitioner.

For a deep learning model we’ll evaluate our model by looking at the result of the loss function, both during training and validation (as described above, this can help determine whether we are under- or over-fitting). We’ll also consider performance metrics such as accuracy, precision and recall, and investigate how they improve with time (with training). If at some point they fail to improve and reach desired levels, then the architecture must be adjusted and the training process begun anew.

Perhaps counter-intuitively the next step is to push the model architecture (e.g. add nodes or layers of nodes) such that the model overfits the data. Without taking this step an important question would stand – is the model actually capturing the full complexity of the data or could it do better with more parameters (more nodes)? A network with a single layer might do better than baseline, but not do as well as it could do. The universal tension in machine learning is between optimization and generalization – the ideal model stands at the border between underfitting and overfitting. To find this border, the model needs to be tuned by crossing it. The researcher can do this by adding more layers, making the layers larger, and training for more epochs. Overfitting is achieved when, as the training time increases, validation loss decrease stops tracking training loss decrease (the model is able to better fit the training data, but that fitting doesn’t generalize to new data).

Once this boundary has been identified, the model can be fine tuned. This takes the most time in the deep learning process – the researcher needs to repeatedly modify the model, train it, and evaluate on the validation data, until the model is as good as it gets. This tuning stage often focuses on hyperparameters, such as the number of units per layer, or the learning rate of the optimizer, but it may also drive the investigator to reevaluate the set of features being investigated (and whether some appear to be irrelevant or others seem to be needed).

The deep learning processes in this thesis have been programmed in Python using a number of packages, the most important being the NumPy\textsuperscript{75} and Keras\textsuperscript{76} packages. Keras is a high level deep learning API that runs on lower-level deep learning packages such as TensorFlow\textsuperscript{77}. It was developed for the researcher to be able to quickly experiment to create models, to be able to go from idea to result with the least possible delay. It allows for easy and fast prototyping, supporting both convolutional and recurrent neural networks, and runs seamlessly on CPU and GPU. The package is analogous to density functional theory packages in that they are high level APIs which require deep knowledge to meaningfully use. An example of the code required to perform the tasks described in this last section can be found in appendix C.

For a more in depth introduction to Deep Learning, I recommend Deep Learning by Ian Goodfellow et al\textsuperscript{41}, and Deep Learning with Python by Francois Chollet\textsuperscript{37} for an introduction to the Keras package in Python.
Chapter 3

Vibration Cancellation in Scanning Probe Microscopy using Deep Learning

The high sensitivity of scanning probe microscopes poses a barrier to their use in noisy environments. Vibrational noise, whether from structural or acoustic sources, can show up as relative motion between the probe tip and sample, which then appears in the probe position (“Z”) feedback as it tries to cancel this motion. Our group, primarily through the efforts of Lavish Pabbi, has developed and patented an active vibration cancellation system designed to take advantage of existing feedback and drive systems in an SPM in order to cancel the effects of vibrations. In this chapter I will discuss an extension to this technique that I pioneered using deep learning.

3.1: Motivation

All scanning probe microscopes (SPMs) are sensitive in some degree to external vibrations, as their measurements depend on maintaining a very small (atomic scale) constant tip-sample separation. Typical efforts to eliminate the effects of vibrations focus on the structural design of the instrument, often by making the tip-sample junction as stiff as possible (typically pushing resonance frequencies into the 1-10 kHz range) while supporting the system on multiple soft-spring isolation stages (with resonance frequencies in the 1-10 Hz range). Even with these efforts, highly sensitive instruments typically require a very quiet lab environment. This makes it difficult to use active refrigeration techniques, like cryocoolers, or to combine the STM into an instrumentation suite with potentially noisy tools. A variety of other vibration cancellation systems have been developed, both for STM and for other vibration sensitive instrumentation, yet none have been widely adopted, likely because of their complexity, expense, or narrow range of use.

Our lab has developed the Active Noise Isolation for Tunneling Applications (ANITA), a system which relies on existing tip positioning technology to stabilize the tip-sample junction, but moves the signal associated with vibrational motion out of the main current/Z-feedback loop by correlating it with accelerometer measurements of vibrations.

Our original algorithm uses a linear transfer function method to train the model to predict the feedback signal. Because other feedback systems have benefitted from implementing machine learning techniques, I decided to implement a recurrent neural network-based algorithm and test for improvements to our vibration cancellation system.
3.2: Basic Experimental Setup of ANITA

Although the details of ANITA can be found in our patent\textsuperscript{28} and paper\textsuperscript{28}, in this section I will briefly describe its operation in order to clarify my machine learning versions, both offline and online. A schematic of ANITA is shown in Figure 3-1. The primary addition to a standard STM setup is a Geophone\textsuperscript{112} (accelerometer) for sensing mechanical vibrations, whose signal we call $G$. Operation of ANITA is a two-step process. After bringing the system into tunneling we first perform a training step (Figure 3-1a). The signal from the geophone, as well as the “ANITA off” STM controller $Z$-feedback ($Z_{FB-}$) signal, are fed, for training, into the ANITA Processor and the system is, as usual, run only using this feedback ($Z_V = 0$). When switched on, ANITA uses a real time digital analysis of $G$ to create a vibration control signal $Z_V$. Adding $Z_V$ to the controller’s $Z_{FB}$ transfers the vibrational portion of the feedback to the ANITA controller, segmenting the relative tip-sample control into vibration ($Z_V$) and remaining feedback signal ($Z_{FB+}$).

One may ask whether using an active feedback system such as ANITA risks “contaminating” the raw data an SPM would usually produce. This concern is, however, unfounded. As the typical SPM feedback continues to run in constant current mode, the sum of this feedback system ($Z_{FB+}$) and the ANITA signal ($Z_V$) will reproduce the signal that would have originally been produced by the STM controller in the absence of the ANITA system ($Z_{FB-}$). We are merely segmenting this signal in order to isolate the non-vibration caused tip motion.

Figure 3-1: ANITA schematic and concept. (a) A typical SPM maintains tip and sample separation via $Z$-feedback ($Z_{FB}$), generated in the controller. ANITA adds a geophone, whose signal $G$ is correlated with $Z_{FB}$ during a training step (dashed line), and then used to generate a $Z$ vibration signal ($Z_V$). Adding $Z_V$ to $Z_{FB}$ transfers the burden of cancelling vibrations from the controller to ANITA. (b) Model segmentation of a topography as it would appear with ANITA vibration cancellation off ($Z_{FB-}$) into the ANITA determined vibration control signal $Z_V$ and a now (ANITA on) “vibration-free” feedback signal $Z_{FB+}$. 
3.3: Linear Transfer Function Model

An essential part of ANITA’s operation is a training algorithm, which leads to a model that can predict $Z_V$ from $G$. The original ANITA model assumes that the relationship between the two is linear and time invariant. Thus a linear transfer function, $H$, determined from training data as:

$$H = \mathcal{F}^{-1}\left[\frac{\mathcal{F}(Z_{FB})}{\mathcal{F}(G)}\right]$$

(3-1)

This is, in usage, simply convolved with the geophone signal to determine the predicted vibration signal $Z_V$:

$$Z_V = H \ast G = \mathcal{F}^{-1}[\mathcal{F}(G)\mathcal{F}(H)]$$

(3-2)

3.4: Recurrent Neural Network Model

This model, however, makes several assumptions about the relationship between the geophone signal, which measures acoustic and mechanical vibrations external to the SPM, and the relative motion of the tip and sample inside of the SPM. Namely, tip-sample motion at any given frequency is assumed to depend only on geophone-measured vibrations at that same frequency, and to do so linearly (if one doubles then so does the other).

Although these assumptions seem to be okay at least as first order approximations (ANITA works quite well) the application of a deep learning model which discards these assumptions allows us to both test them and potentially improve on the success of the linear model.

In determining which machine learning model to use, it is important to note that although scanning can be thought of as producing spatial data, the $G$ and $Z_{FB}$ signals are simply time series and inherently sequential. As explained in section 2.4.3, recurrent neural networks (RNNs) are an appropriate model to use to generate predictions from sequential data. A recurrent neural network does not make any assumptions as to the structure of the data being fed as input for prediction. In this model, we feed a sequence of $G_{t-h}$ of a certain window size $h$ to an RNN. This returns a prediction $Z_t$ for each sequence. Much like the linear transfer function algorithm, for each later $Z_t$ point, the $G$ signal being fed into the

![Diagram of RNN](image)

Figure 3-2: Model of the RNN used to predict $Z_{FB}$ from $G$. A sequence of $G_t$ points are fed into an RNN unit, which has memory, and the very last RNN unit spits out a single $Z_{FB}$ prediction – “many to one”.
algorithm is shifted, appending the newest data point and dropping the oldest. This is an example of a “many-to-one” RNN, predicting a single value from a series of values, as seen in Figure 3-2.

Here, I've implemented a Gated Recurrent Unit (GRU), a model similar to but simpler than the Long Term Memory Unit (LSTM), and which has been shown to exhibit better performance on small datasets. GRUs also have fewer parameters than LSTMs, and train faster. The architecture used for the RNN is a single GRU layer between the \( G_t \) input and a single neuron dense layer with no activation for the \( Z_t \) prediction. A sequence of \( h = 400 \) points, corresponding to a window size of 1 second (due to the sample rate of 400 Hz) is used to prime the RNN. \( Z_t \) predictions are preprocessed by applying a 1 Hz butterworth high-pass filter. Both \( Z_t \) and \( G \) are standardized before being fed into the RNN by subtracting the mean and dividing by the standard deviation of the signal in the training set. This feature scaling (as described in section 2.4.4) is done to allow gradient descent to converge faster. This standardization is inverted in later analysis. The model was trained in Python with the Jupyter Notebook system using the Keras package for deep learning, with a Tensorflow backend. Models were saved using Keras’ abilities to save HDF5 files with H5Py.

To directly compare this model with the linear transfer function model, we first investigated the nature of the \( G \) and \( Z_{FB} \) signal data by performing an exploratory analysis of the time series and their frequency spectrums to look at the relationships between the two signals. We then apply the two algorithms: we train with the linear transfer function method on a window in data located in the training set. Similarly, using the RNN model, we train the algorithm, holding back a portion of the training set for validation. We then use both models to predict \( Z_t \) on the test set. Details about the exploratory analysis of the data are in section 3.5, while the comparative results of the predictions of both models are seen in section 3.6.

3.5: Exploratory Analysis of Time Series

To explore the nature of the relationship between the geophone and Z-feedback time series, we repeated the experiments discussed in our publication. We drove the system using a single frequency vibrational source — a dynamically unbalanced, mass-loaded fan mounted near the STM chamber. We can vary the frequency of vibrations by tuning the loaded motor DC drive voltage and the amplitude by varying the load mass; here we have tuned them so the vibration is clearly observable without damaging the tip. We made measurements of the \( G \) and \( Z_{FB} \) signals (labelled \( Z \) in this section for simplicity) at room temperature with a Pt-Ir tip on a gold sample in constant current feedback. We took two sets of measurements: A training set 320 seconds long and a test set 128 second long. Both series of measurements had the same sampling frequency of 400 Hz. After the measurements, we used Python to perform an exploratory analysis of the \( G \) and \( Z \) signals of the training set.

3.5.1: Time Domain Analysis

The first step in investigating the two time series is to see how they evolve in the time domain, both at long time and short time scales. Figure 3-3 shows the long time trend of the \( Z \) series. Figure 3-3(a) shows that \( Z \) is not stationary — there is a long term drift, most likely caused by thermal drift of the tip. This
drift term can be removed by applying a 1 Hz butterworth high-pass filter to make the signal stationary, as seen in Figure 3-3(b). The amplitudes of the vibrational signal are on the order of 0.10 nm.

The geophone signal, by contrast, is stationary as it only measures the external mechanical vibrations outside of the STM, and it lacks a thermal drift mechanism. Figure 3-4 shows the short term trends of the Z and G signals. The Z is the filtered stationary signal. Both of these signals have a similar dominant periodic signal of 17 Hz, generated by the mass-loaded fan.

3.5.2: Frequency Domain Analysis

As both time series are periodic, the most natural place to explore the data is in the frequency domain. I performed spectral analysis by taking spectrograms and global spectral densities of the G and Z signals. This allows us to see how the spectral components of the signals are changing as a function of time.

Figure 3-4: Short term trend of Z and G signals. Both show a dominant periodic signal of 17 Hz, created by the mass-loaded fan. The geophone signal shows a protrusion near the turning points, not seen in the Z vibration signal.
The most important point from these graphs is that there are time-invariant peaks in the frequency domain corresponding to periodic vibrations. The strongest of these peaks is at 17 Hz. The $Z$ signal also has sporadic broadband noise, showing as time-varying fluctuations in the frequency domain. Although variations in $G$ are not nearly as obvious as variations in $Z$, it will be interesting to see whether there are any signatures that can be used to predict this broad-band noise – something that certainly couldn’t be done with the original linear model given the lack of comparable signal in $G$. If the broadband noise were associated with user actions, such opening/closing doors or typing on the keyboard, then it would seem likely that we would see the noise in $G$ as well. As the broadband noise is not being picked up in the geophone, it’s possible this is noise that is internally generated (such as from boiling liquid nitrogen) that the geophone in its current position may not be sensitive to. In addition, the STM tip is more sensitive than the geophone and can pick up more subtle signals. These time varying-components show up as fluctuations in the noise floor of the $Z$-spectrogram, rather than fluctuations in the vibrational peaks.

Figure 3-6 shows the global spectral densities (across the entire time series) of the $G$ and $Z$ signals. Most of the vibrational peaks in the geophone signal match those of the $Z$ signal. The most prominent peak is the frequency of the fan, at 17 Hz. The noise floor is different in both signals. The geophone signal shows a flat, white noise background. This is most likely due to Johnson noise, which arises from fluctuations in voltage across a resistor. In contrast, the $Z$ signal shows a background that decreases as the frequency increases, roughly following a pink noise (1/f) fall-off. Most, but not all, of the peaks in $Z$ have a matching peak in $G$ (at the same frequency) – this was one of the assumptions of the linear transfer function algorithm.
3.5.3: Cross-Correlation Analysis

Two ways of representing the correlation between two signals are the cross-spectrum (or cross spectral density), which highlights common spectral peaks in two different time signals, and the squared coherence, which indicates how well correlated two time signals are as a function of frequency (see Appendix A for details). These are shown for Z and G in Figure 3-7 (a) and (b) respectively. The cross-spectrum shows vibrational peaks similar to those observed in the individual Z and G signals, most prominently near 17 Hz. At these same vibrational frequencies, the coherence reaches a value between 0.1 to 1 (where 1 would indicate perfect ability to predict one signal from the other at that frequency).

Figure 3-6: Global spectral densities of G and Z signals. Vibrational peaks in both signals match, with the largest peak in both signals at 17 Hz, corresponding to the fan frequency. Other peaks in the G signal seem to have correspondences in Z.

Figure 3-7: Cross-Spectrum and Coherence between the Z and G signals. (a) Cross Spectral Density. Vibrational peaks prominent in both signals appear here. (b) Squared Coherence, a time-series analogue of Pearson Correlation. The vibrational peaks in both signals approach 0.1 – 1.0, while the noise floor isn’t as well correlated at ~10^{-2}.
The noise floors (in between spectral peaks) show a coherence of around $10^{-2}$, meaning, unsurprisingly, that at these frequencies the (small) signals aren’t mutually predictive. This analysis suggests another possible approach to selecting vibrational frequencies for the conventional ANITA model. Instead of selecting specific commensurate or incommensurate frequencies for modeling, a researcher can choose frequencies with a coherence of 0.1 or larger, where the model is likely to perform better. This analysis also shows the limitations of predicting $Z$ from $G$ as the coherence is not close to 1 at all frequencies in this band.

3.6: Comparative Results of Predictive Models

To compare the linear transfer function (LTF) model with the recurrent neural network (RNN) model, we performed predictive analyses of both models offline in Python. The LTF model was performed for direct comparison, as the RNN model is only trained once. Both models were trained on a training set; the LTF model was trained on a window of the training set to simulate how the system currently works, while the RNN model was trained on the entire training set.

![Model Performance of the LTF and RNN models.](image)

Figure 3-8: Model Performance of the LTF and RNN models. Each graph has three time series: (i) True $Z$, representing the feedback before separation. (ii) Predicted $Z$, which represents the vibrations being removed from the signal. (iii) Error, which represent the vibrations that remain in the feedback.

To measure performance, we predicted the $Z$ signal on a test set separate from the training set we used to create the model. Figure 3-8 compares the $Z$ signal prediction success for the the LTF and RNN models. Subtracting the prediction from the true value simulates the effect of adding the vibration back into the feedback signal in the active cancellation regime. The RNN’s error signal has a visually smaller amplitude compared to the LTF model.

<table>
<thead>
<tr>
<th></th>
<th>LTF</th>
<th>RNN</th>
<th>Further Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>190.5</td>
<td>104.4</td>
<td>45.2%</td>
</tr>
</tbody>
</table>
To quantify the performance of the models as a whole, we can calculate the mean squared error, a typical metric in machine learning for regression problems (Table 3.1). While the LTF model has a decent mean squared error, the RNN model performs even better, further reducing the mean squared error by 45.2%.

Figure 3-9 shows the spectral densities of the original test set of $Z$ before correction, and the reduced signals produced by the LTF and RNN models. The RNN model shows more peak reduction in the main frequency at 17 Hz, and more reduction at higher frequencies. The pink noise background remains in both algorithms, showing that it is difficult to remove this even in a black box context. Investigating the relative noise reduction of the 17 Hz peak, the RNN model is better than the present ANITA model by roughly 3.4 times.

3.7: Summary

The patented ANITA algorithm uses a linear transfer function algorithm to make predictions of the Z-feedback vibrations from a geophone signal measuring external vibrations. I have performed a time series exploratory analysis to understand the relationships between the geophone signal and the vibrational signal, and have created a nonlinear deep learning model that I hypothesized would perform better than the transfer function method, and have reduced the error in the signal by nearly 50%. The largest peak in the signal has also been reduced by a factor of over 3 times compared to the linear transfer function method.

There are two directions in which this project can move forward in – creating an active, “online” system or a post-processing, “offline” system. Each approach has its challenges and potential benefits.

Integrating the deep learning model with the active system, in which we add the vibrational predictions directly into the feedback signal in the STM controller, will allow us to directly improve the vibration...
cancellation process from the linear transfer model, as seen in section 3.6 by the increased performance (reduced error signal). There are two challenges to this approach. First, the training process for the deep learning model is extremely slow – depending on the complexity of the model, training can last from 30 minutes to hours (compared to a few seconds for the patented ANITA algorithm). Training a model has to be done offline, separate from the active system, before being integrated into the online system. This wouldn’t necessarily be an issue, as in deep learning making a prediction is far faster than training a model. But for an SPM user anxious to get on taking data it could prove problematic. Furthermore, the prediction process, while fast, is not currently fast enough to be integrated into the feedback system (it can run at about 100 Hz but would need to be at least 1 kHz to keep up with the system sampling frequency). The deep learning model was created with the python package Keras, which isn’t well suited for real time predictions due to the overhead inherent in Python. To improve performance we could consider reimplementing the code in C or C++. Having done this, it may be possible to fit our deep learning prediction model in a CPU as simple as a Raspberry Pi, as was done in the RNNoise active noise cancellation project.

Another approach would be to abandon real time processing (and the speed demands associated with it), and instead integrate the deep learning model into a post-processing offline system. Here, we take the geophone signal and noisy image data and run the vibration cancellation step after collecting the data. While a number of post-processing routines are currently used by researchers in the field, such as fourier filtering, the advantage of using an offline vibration cancellation model is that with image data alone, you don’t necessarily know which part of the image is signal or noise – when filtering out potential noise signal you might be filtering out real data too. The additional physical information provided by the geophone and incorporated in an “offline ANITA” system can reduce this possibility. The process here would be to record the geophone data along with any typically recorded signals, then to “unwrap” the data into a set of parallel one-dimensional time series, train a deep learning model, predict the vibrations and subtract the predicted vibrations to obtain a “noiseless” image.

In addition to the benefit of reducing speed requirements, this could also potentially enable usage in parallel with another active cancellation system, or post-data collection model training to continuously improve results. Unfortunately the offline technique also has some challenges, which we have thus far been unable to overcome. In order to work properly, time series analysis such as I have done here must be performed on data equally spaced in time. This provides a challenge because unlike oscilloscope collected time series (which was the source of the data for this analysis), scan data tends to have delays at various points in the image (for example, at the end of scan lines) so that the time spacing between individual pixels isn’t fixed. Even if we managed to time stamp the pixels (which we attempted using various methods, including, for example, recording the voltage of a ramp function produced by a precision function generator so that the voltage could be linked to time), because the time delay between certain pixels (again, for example, at the end of a scan line) isn’t necessarily an integral multiple of the typical time spacing between pixels, there is no good way for the RNN model to handle these random chunks of “missing data” (times with no input).

The results discussed in this chapter are novel, and I presented them at the 2018 APS March Meeting. However, because we have not yet come to an operational model, we have not published them. Both approaches await further development by the next generation of SPM data scientists in our group.
Exploiting big data techniques in scanning probe microscopy requires the construction of a massive database of experimental data. Metadata tied to the experimental data can be either obtained directly from experimental conditions or manually annotated using expert knowledge. This annotation process can be accelerated using deep learning to automate the creation of metadata. We use a convolutional neural network (CNN) to classify STM topography scans on atomic resolution and image quality. We have achieved an accuracy of roughly 90% for atomic resolution and 80% for image quality. The creation of this system lays the foundation for enabling automation of the SPM data collection process.

4.1: Motivation

The first step to exploit big data techniques in scanning probe microscopy (SPM) is to construct a database of experimental data. In addition to the SPM data itself, the database should also include associated metadata describing the experimental situation – sample investigated, tip used, temperature and magnetic field, and so forth. Some of this data (such as temperature) is recorded automatically during data acquisition, and other data (such as sample & tip information) is recorded prior to data acquisition. However, it is often advantageous to further annotate the data after acquisition.

One example of this being useful is when a new analysis pathway opens up and the researcher wants to comb through old data, looking for feature X. Given that SPM groups can produce thousands of topographies and millions of spectra a year, it isn’t practical for a researcher to go back and manually tag this old data. In our group the practice is to keep extensive records of “high quality datasets” which we think may be suitable for mining in the future. But even with this, the question of “do we have any data that shows X” frequently arises and is challenging to answer without some automated feature search.

A second benefit of automated data annotation is the possibility of automated (or guided) acquisition. If the computer were able to search for a desired feature in data, and know how (typically) to get from current acquisition conditions to those that show that desired feature, it could either advise the operator or, potentially, drive the acquisition itself.

As a first step in this process, I decided to create a deep learning model to search for two features in STM topographs: the presence of atomic resolution and high quality data. Deep learning, in particular with convolutional neural networks (CNNs), has already been used with microscopy data to perform a wide variety of tasks, such as enhancing the spatial resolution of optical microscopy\textsuperscript{25}, detecting atomic defects in STEM scans and extracting relevant physical and chemical information\textsuperscript{24}, automatically detecting and reconditioning a scanning tunneling microscope tip\textsuperscript{26}, and labeling features in scanning electron microscopy data\textsuperscript{23}. The search for atomic resolution in STM topographs is quite similar to several of the studies mentioned above. However, while there is literature on the classification of objects observed in microscopy data, there haven’t been projects focussed on assessing the quality of
STM images. This is necessarily a challenge due to the subjective nature of image quality. However, differentiating image quality is a skill that STM experts must develop in order to effectively take data so something that the machine must also learn in order to either advise or drive the acquisition process.

For this project I trained and supervised an REU student, Kevin Crust. I worked with him on all aspects of the project, though some aspects (for example, the initial idea) were primarily or wholly mine, while others (like manual annotation of the training and testing data) were primarily his.

4.2: Data Collection and Annotation

To begin the classification process, we gathered together a series of over three thousand STM topography scans of various quality. The scans were taken from different runs of our STM over multiple days while investigating five different material systems. We intentionally chose files that were a mix between good and bad images, reflecting the natural diversity of image quality obtained in our SPM system.

To process the scans, we first converted the SXM format (used by our Nanonis SPM Control System\textsuperscript{119}) into two separate formats: PNG picture files for human visualization, and array data stored in the HDF5 file format\textsuperscript{120} for computer analysis. All of the images were square, but of different pixel sizes and scan ranges. Samples that were less than 512 x 512 pixels large were first interpolated to 512 x 512 using a fourier interpolation method. Images larger than this were cropped into four 512 x 512 sized samples. All of the topographies had a linear background subtracted. The binary files were later downsampled to 256 x 256 to feed into the deep learning model. Including the cropped images, there were a total of 4542 samples.

Kevin used an online manual image annotator, Dataturks\textsuperscript{121}, to label the data for training and testing. This tool, free for academic researchers, allows one to fairly quickly annotate images for image classification and other purposes.

The images were annotated into three different categories: type of material, existence of atomic resolution, and image quality. The type of material was known in advance. The dataset consists of five different materials: boron nitride doped graphene, nitrogen doped graphene, calcium doped Bi\textsubscript{2}Se\textsubscript{3}, chromium doped Bi\textsubscript{2}(Sb\textsubscript{1-x}Te\textsubscript{x})\textsubscript{3}, and WS\textsubscript{2}. The type of material is not currently used for prediction, as the classes are highly unbalanced, with two thirds of the samples being BN-doped graphene.

Image quality was divided into three different classes: poor, fair, and good quality. We note that these are inherently non-quantitative categories, but something that STM experts generally agree on. When training Kevin for the manual tagging process, we had meetings with all members of the group comment on image quality and there was always consensus. Kevin also skipped questionable images on his first tagging pass, and consensus was also reached on these more challenging images. Future efforts along these lines should involve multiple taggers to determine interrater reliability, which would necessarily limit computer rating accuracy. In essence, the categories may be described as follows. Poor quality images are typically those with lots of noise, or images that are incomplete. They are images from which it is impossible (or exceedingly difficult) to extract scientifically useful information. Good quality images are clear images that are of publishable quality, and are the smallest part (7%) of the dataset. Fair
quality images are those images with some level of noise but not to the level of low quality images, are more scientifically useful than poor quality images. In terms of guiding acquisition, good quality images are the end goal, fair quality images typically contain enough information to point in the direction of good images, while poor quality images often give little to no guidance. Figure 4-1 shows examples of topographies of varying resolution image quality.

![Example images of STM topographies](image)

Figure 4-1: Examples of STM topographies of different resolution and image quality. Poor quality images typically have disruptive levels of noise and can be present in both atomic and non-atomic resolution. Good quality images have much clearer features. Fair quality images are in between.

Table 4-1 shows the distribution of the classes of different categories in our dataset. There is a correlation between atomic resolution and image quality (with a moderate polychoric correlation of 0.620 ± 0.014 – see Appendix A for details). This is unsurprising – the existence of atomic resolution is often challenging to detect if the image quality is not at least fair. The dataset is completely dominated by BN-doped graphene, and for this reason we decided to not pursue the material metadata.

Table 4-1: Annotations of our STM Topography Dataset. Left section is atomic quality vs. resolution; right section is the number of images per type of material. There were a total of 4542 images after dividing and cropping the scans.

<table>
<thead>
<tr>
<th>Material</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN-Graphene</td>
<td>3058</td>
</tr>
<tr>
<td>Ca-Bi$_2$Se$_3$</td>
<td>463</td>
</tr>
<tr>
<td>Cr-Bi$<em>2$(Sb$</em>{1-x}$Te$_x$)$_3$</td>
<td>193</td>
</tr>
<tr>
<td>N-Graphene</td>
<td>193</td>
</tr>
<tr>
<td>WS$_2$</td>
<td>173</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>non-atomic</th>
<th>poor</th>
<th>fair</th>
<th>good</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>1367</td>
<td>314</td>
<td>100</td>
<td>1781</td>
</tr>
<tr>
<td>Total</td>
<td>1936</td>
<td>2284</td>
<td>332</td>
<td>4542</td>
</tr>
</tbody>
</table>
In addition to the image data itself, the data files each contain additional metadata about the scan. This includes the scan range of the image (important as the image itself doesn’t contain information about its size), sample bias and current set point, scan speed, and PI controller information describing the STM feedback system. For preliminary investigation of the importance of these metadata, we calculate their Spearman correlations (see Appendix A) with atomic resolution and image quality (Figure 4-2), and make the following findings:

- **Scan range and time parameters correlate strongly with atomic resolution ($|\rho_S| > 0.5$).** These correlations make sense – a smaller scan range is often required to see atomic resolution, while we typically scan slower for larger fields of view, where we are less likely to see atomic resolution.

- **Current setpoint correlates moderately with atomic resolution ($|\rho_S| \sim 0.3$).** We typically start with a low current setpoint when we first start taking data. When the current setpoint is lower, the tip is further away from the sample, preventing us from crashing into the sample. At this point,

![Spearman Correlation of Metadata](image_url)

**Figure 4-2:** Spearman Correlation of Metadata with atomic resolution and image quality. Scan Range and Scan Time are correlated with atomic resolution with $\rho_S > 0.5$. The current setpoint has a correlation > 0.25 with atomic resolution. Histograms show the distribution of values for some of the key metadata.
when we are first exploring the sample, we tend to take large scale images, and are less likely to have atomic resolution. Once a quality region with the potential for atomic resolution is identified, we typically increase the current set point (push the tip closer to the sample) to enhance contrast, and, we hope, achieve better atomic resolution.

- All other metadata had less significant levels of correlation with atomic resolution, and no metadata had a significant correlation with image quality.

4.3: Deep Learning Model

Figure 4-3: Deep learning model architecture. Image data is fed into a CNN, composed of four blocks of double convolutional layers, a max pooling layer, and a dropout layer. A metadata vector is fed into a fully connected dense layer. These submodels are concatenated together, fed into a final dense layer, then split into the two different outputs – a binary atomic classifier, and a multiclass quality classifier.

4.3.1: Architecture

The deep learning model architecture that has been designed is a multi-input, multi-output model. Each sample fed into the model consists of the topographic image data, as well as a vector composed of the metadata we determined above were relatively well correlated to atomic resolution: scan range, scan time, and current setpoint, as well as two feedback parameters: the proportional and integral gains. The image data is fed into a CNN composed of four blocks of four layers – two convolutional layers, a max pooling layer, and a dropout layer. The metadata is fed into a simpler single fully connected hidden layer. These submodels are merged together, fed into another fully connected layer, and then branched into the two outputs: a binary “atomic” output, and a multiclass “quality” output. Figure 4-3 shows a visual diagram of the architecture.
There are two loss functions used in the model. The atomic output uses a binary cross-entropy loss, a loss function typically used for binary classification. The quality output uses a categorical cross-entropy loss, typically used for multiclass classification. The total loss is a weighted sum of these two in a ratio of 5:3 atomic:quality, where we determined the appropriate weighting factors by training a model and noting the relative loss contributions.

Before being fed into the model, the image data was preprocessed. All image data samples were feature scaled (see Sec. 2.4.4): for each topograph we subtracted the mean height and divided by the standard deviation. While we lose information in this process – the model essentially measures contrast rather than topography – as is often the case, the model trained much better after scaling.

In addition, training data was augmented using a number of transformations: shifted or flipped vertically or horizontally. We opted against 90 degree image rotation, as poor quality images are often overwhelmed by horizontal scan lines (see Figure 4-1), and the vertical lines that would appear in poor rotated images are not characteristic of actual STM scan data.

### 4.3.2: Training Process and Hyperparameter Tuning

After building the final model, we trained it for 100 epochs. The first step of the training process is making sure that the model is correctly learning the true labels. We can measure this by plotting how the loss function and accuracy of the model changes as a function of time, measured by the number of epochs that the model has been trained. This is shown in figure 4-4. The loss in the atomic function is typically lower than that in the image quality. After each run through the entire dataset (an epoch), predictions are made on the validation data and the performance measured. Two features are clear in the figure. First, around 20 epochs there is a jump in performance. This is caused by the fact that we are using learning rate decay – when the model starts to decrease performance over a certain number of epochs, we decrease the learning rate of the optimizer by an order of magnitude to improve the optimization, allowing us to better approach the local minimum. Second, around 70 epochs the validation loss and accuracies start flattening out while the training loss and accuracies continue to improve. This is a signature of overfitting – the model ceases to generalize well. The final model is chosen to be the one just before overfitting begins.

There are a number of hyperparameters in the model that need to be tuned. Inside the model, there are parameters within the individual layers, such as the amount of dropout in the dropout layers, the number of filters and the kernel size in the convolution layers, and the number of neurons in the dense layers. In addition there are architecture degrees of freedom – how many convolution blocks, or series of convolution layers followed by pooling? Outside of the model, a proper learning rate for the optimizer needs to be set. All of these hyperparameters affect the performance of the model, so there is a need to find the right values. Below we describe how this tuning is done.
We began by deciding which hyperparameters to tune. After a brief search across each hyperparameter’s space individually to determine the model’s sensitivity to the parameter and the parameter’s likely optimization range, we chose to focus on three parameters: the learning rate on the Adam optimizing algorithm, the dropout rate on all layers except the last, and the number of convolution blocks. Next, we trained multiple sets of hyperparameters for the same number of epochs (40). We used a random search validation process in which we randomly selected values for the dropout and learning rate. Figure 4-5 shows a graphical representation of this analysis. The size of the circles is proportional to the accuracy on the validation dataset.
Because of the potential for interplay between parameters, there is a natural desire to search across a wide number of parameters, and across a wide range for each. However, the several hours required per datapoint pictured in Figure 4-5 limits the number of such calculations. From the trends in this analysis, we determined that good performance could be achieved with a learning rate of $2.5 \times 10^{-4}$, four convolutional blocks in the CNN, and no dropout at the end of each convolutional block except the last. The latter was surprising – dropout (randomly shutting down a fraction of neurons in the layer in each pass) is often required to prevent overfitting. However, it appears that the data augmentation we used, which also helps to avoid overfitting, in combination with the last block dropout was sufficient.

### 4.4: Classification Results

There are a number of ways to investigate the performance of the classification model. All of these ways involve evaluating the model on a separate test set that the model hasn’t seen before, and comparing the predictions with the actual labels.

Table 4-2: Confusion Matrices for the final model.

<table>
<thead>
<tr>
<th>True</th>
<th>Predicted</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>non-atomic</td>
<td>atomic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-atomic</td>
<td>619</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atomic</td>
<td>70</td>
<td>1033</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True</th>
<th></th>
<th>Predicted</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>poor</td>
<td>fair</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>poor</td>
<td>617</td>
<td>145</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>fair</td>
<td>142</td>
<td>757</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>4</td>
<td>59</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-2 shows the confusion matrices for the model on the 1817 image test set. Ideally, all of the items should be on the diagonal – predictions should match actual labels. The lack of balance along the
diagonals reflects the lack of balance in the actual labels—the number of atomic labels is greater than
the number of non-atomic labels, for example, and there are many fewer good images than either fair
or poor. A good sanity check of our model is the fact that there are few poor images that are predicted
to be good, and vice-versa. The overall accuracy of our model is reasonably high—90.9% for atomic
resolution and 79.1% for image quality. For comparison, a “dummy” classification model which
randomly selected a label regardless of the underlying data would achieve 53.0% accuracy for atomic
resolution, and 22.7% accuracy for image quality. Thus, the model has statistical power, providing
results that are better than random chance. It also makes sense that the accuracy is lower for image
quality compared to atomic resolution, as it is comparing results for three labels compared to two
labels.

However, accuracy doesn’t always provide a full picture of model success. For example, if 90% of the
data is a single label, then the model can achieve 90% accuracy by simply labelling all data as that single
label. Thus, due to the imbalanced nature of our dataset, and given that we did the hyperparameter
search based on accuracy, it is important to investigate the category specific metrics of precision and
recall (Table 4-3).

The precision and recall of the model for atomic resolution are in line with the high accuracy, with 90.9%
mean precision and recall for the model. For comparison, a dummy classifier achieves 52.9% mean
precision and 53.0% mean recall.

However, the image quality metrics tell a different story. While poor and fair precision and recall are
relatively high (around 80% for both, similar to the overall accuracy), the precision and recall for the
good label are lower. They are still greater than chance—the precision and recall for a dummy classifier
on good images are 5.0% and 4.8% respectively. But it is troubling, as the 50% recall means that the
model only correctly predicts half the actual good labels (the other half it mostly predicts as fair) while
the 68% precision means that only two-thirds of the images it predicts as good are (the other third are
nearly all fair). There are a couple of possible reasons for this. First, the proportion of good labels in the
training set is low—only 7.3% of the data. In machine learning, label prediction performance tends to
decrease as the label proportion decreases. However, another possible issue is the qualitative nature of
the labelling. To the extent that Kevin had difficulty choosing labels in some cases, even though in the
end we did come to a consensus decision on those labels, his lower confidence in choosing those labels
originally is an indication of the difficulty of the task.

Table 4-3: Performance metrics of the classification model

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F$_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-atomic</td>
<td>89.8%</td>
<td>86.7%</td>
<td>88.2%</td>
</tr>
<tr>
<td>atomic</td>
<td>91.6%</td>
<td>93.7%</td>
<td>92.6%</td>
</tr>
<tr>
<td>mean</td>
<td>90.9%</td>
<td>90.9%</td>
<td>90.9%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F$_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>poor</td>
<td>80.9%</td>
<td>80.8%</td>
<td>80.8%</td>
</tr>
<tr>
<td>fair</td>
<td>78.8%</td>
<td>81.7%</td>
<td>80.2%</td>
</tr>
<tr>
<td>good</td>
<td>67.7%</td>
<td>50.0%</td>
<td>57.5%</td>
</tr>
<tr>
<td>mean</td>
<td>78.9%</td>
<td>79.1%</td>
<td>78.9%</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>90.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quality</td>
<td>79.1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To look into this latter possibility, we investigated the model’s classification confidence, which it outputs for each sample as a probability of each possible classification label being correct. Thus far we have focused exclusively on the label with the highest probability, ignoring the probability itself. However, a histogram of confidence levels, separated by correct and incorrect predictions (Fig. 4-6) highlights an important model trait. If we consider above 70% confidence to be “high confidence,” then in the cases that the model is highly confident, it achieves 95% correct atomic resolution labelling on 87% of the data, and 87% correct image quality labelling on 79% of the data. That is, the model is highly confident in labelling most of the data (87%/79%), and, when confident, is usually correct (95%/87%), while when not is nearly random (about 58% for both tasks). Depending on the goal of the automatic classification, we could use this information in several ways. For example, we could only automatically classify those with a high confidence and manually label the remaining 10%-20% of the samples.

4.5: Summary

We have classified the presence of atomic resolution and the image quality of various STM topographic scans and created a model which successfully predicts these labels. We achieved a model with statistical power, achieving roughly 90% accuracy on atomic resolution and 80% on image quality. The model predicts poor and fair quality images quite well, but doesn’t predict good quality images nearly as well.
We can improve the performance of the model further using two different strategies. First, we could add more good quality images to the training set – off-balance labels tend to lead to decreased performance. Improving the balance by adding more good images will help the model learn how to better predict good quality images. Second, we could add more images of topographic scans of different types of materials. This could potentially help generalize what is considered “good quality” in a material independent fashion. Both of these strategies would require the tagging of more data, which may seem excessive, given that we have already tagged over 4000 scans. However, “big data” datasets for training deep models typically consist of millions of samples. For example, ImageNet, a large visual database used to train visual recognition software, uses 14 million images trained on 1000 different classes. Thus it is reasonable to consider more data. In our group alone we have at least ten times more data we chose not to include (in the interest of time), and collaboration with other research groups could increase the pool of data even further.

However, before working on improving the model, it will be important to investigate a natural limiting factor on accuracy, precision and recall – the quality of human annotation. Even determining the presence of atomic resolution can be non-trivial. For example, in fair and poor images it can be difficult to distinguish atomic resolution from periodic noise. Image quality, as a subjective measure, is even more challenging. As a first step, we should investigate the incorrect ML predictions, and see whether or not they would be better classified than they were when originally annotated. The next step is to investigate interrater reliability. Although as discussed above we had group discussions about a small subset of the data and reached consensus, one thing that we didn’t do was have multiple people annotate the data individually, and determine the level of agreement in the absence of discussion. Seeing how humans agree on image quality annotation could provide bounds to the further improvement of the model.

Finally, as mentioned in the motivation section of this chapter, an ultimate goal of this research is to create a system which could either advise or self-drive data acquisition in search of particular features (such as image quality). Some very preliminary work in this regard has been done by others. For example, investigators developed a genetic algorithm to optimize scan parameters. But this work, performed before the introduction of deep learning, uses a roughness metric to define image quality, which we found doesn’t necessarily correlate well with image quality (for example, the highly stepped image in the upper right of Fig. 4.1 is “rough” but “good” by our metric). Our approach, growing out of the efforts described in this chapter, is fundamentally different. Instead of attacking every possible parameter (as listed in highlighted in Fig 4-2) individually, instead we will train a model based on what experts chose to do overall in a given situation, and on what the results of their actions were. With the ability to decide whether an image has atomic resolution, for example, we can now investigate time sequences of data, noting parameter changes from one to the next, and seeing what, if anything, was done that led to a change from non-atomic resolution to atomic resolution. Or, similarly, the machine can search for changes from poor to good quality data, and see what was done to make that happen. Clearly this is a challenging problem, but one well worth pursuing.
Chapter 5

DataView: Advanced Analytics Software for Multidimensional Data

A key theme in this dissertation is the development of new ways of analyzing SPM data. A number of software packages already exist to visualize and analyze such data (see Appendix D). However, they tend to suffer one of two problems. Either they are GUI-based, high level and hence relatively easy to use software but with a lack of flexibility to deal with the wide variety of data and analysis routines associated with SPM, or they are lower level, command-line driven packages, which are challenging for the novice to navigate. We have developed DataView in order to strike a balance between these two, with an easy to use graphical interface that a novice can quickly navigate, but the power to display, process and analyze a wide array of data, and with a library of routines that have been found to be useful for SPM, but the ability to quickly create new routines as plug-ins.

5.1: Motivation & Existing Packages

In considering the use of existing SPM data analysis systems, commercial packages perhaps have the most problems. First, many are connected to hardware-specific data collection systems, but beyond that, most are also very expensive, not open source, lack advanced or user-defined data analysis routines, and store experimental data in proprietary formats\textsuperscript{119,124,125}. Over the past two decades, however, a number of open-source SPM software packages have been developed. Some examples that are designed to be easy to use include Gwyddion\textsuperscript{126}, WSXM\textsuperscript{127}, and GXSM\textsuperscript{128}. Each of these packages has their own advantages. For example, GXSM and WSXM double as both data acquisition and data processing software. Gwyddion provides a number of sophisticated topographic analysis algorithms and supports many different data formats. However, these packages are still rigid in the type of data they are meant to analyze, in particular only two or three dimensional data, and are not designed to handle large multidimensional datasets.

Pycroscopy\textsuperscript{129}, developed during the time that I have been developing DataView, is a step in the right direction, having this desired functionality in a flexible software package. However, it is command line driven, unlike the GUI-based SPM analysis software mentioned above, and thus inherently has a steep learning curve. We believe that a modern data analysis package for SPM should be able to flexibly view and analyze multidimensional sets within a GUI environment, so that while its users have the ability to easily extend the program, to develop specialized algorithms when needed, at the same time they shouldn’t need to think about programming if they don’t care to.

Although we had a number of other requirements for the software, which will be discussed throughout this chapter, one, we think, stands out as a general deficiency of all existing products we have reviewed, though especially GUI-based products. When performing data analysis, we believe it is critical to automatically log the steps taken by the user, in detail, so that the analysis can be easily investigated and replicated.
In the remainder of this chapter, I will provide a broad overview of some important features of DataView from both a user and programmer’s perspective. Because our group often uses theses as “how-to guides,” a more detailed description, with information about installing and using the software and details designed to allow programmers to jump in and write extensive additions to the program, are provided in Appendix E. And code samples highlighting some of my programming philosophy (which is directly related to the usability of the software from a programmers perspective) are provided in Appendix F.

5.2: History

DataView is an open-source, flexible, multidimensional data analysis software package in Python whose goal is to visualize, analyze, process and catalog a wide variety of data. Although its primary data target is Scanning Tunneling Microscopy (STM) data, it is designed to be completely customizable with the easy ability to add plug-in processing and analysis methods, file handlers, and viewers.

DataView is a descendent of software developed at NIST. The first program was ImageView, developed in the 1990s for data analysis of Scanning Tunneling Microscopy (STM) and Scanning Electron Microscopy with Polarization Analysis (SEMPA). This was upgraded to new software called NISTView using the IDL programming language in 2000. Eric Hudson took control of this project and collaborated on its continued development with researchers at NIST, Harvard, Berkeley and Cornell.

However, in the 2010s, several issues brought to the fore long standing problems with NISTView. An IDL upgrade broke large segments of the code, and IDL itself, as a proprietary programming language, was cost prohibitive for several investigators and counter to the open source spirit of NISTView. Furthermore, the software, like those mentioned above, had been written with very specific data structures in mind – 1D, 2D and 3D data for spectra, topographies and spectral surveys respectively. Though hacks had been made to the code to try to increase flexibility, adding any new analysis modules required extensive editing in multiple parts of the code, as well as frustrating duplication, as analysis of stand-alone 1D spectra vs. 1D spectra embedded in a 3D survey had to be handled differently.

Thus it was decided that the core of NISTView would be scrapped and a new, more flexible and easier to use and program version developed from the ground up. Python was chosen as a coding language for a variety of reasons, though primarily due to ease of use, the existence of a strong scientific computing back-end, Numpy\textsuperscript{75}, and the existence of extensive existing image processing, analysis and machine learning libraries that we planned to incorporate.

At a collaboration meeting in May 2013 it was decided that I would be the sole developer for the core code in order to have a unified vision for the software. The other collaborators were to work on identifying plug-ins (the way we handle all functionality visible to the user – displays, methods, file handlers, etc) and, once the core code was developed, to translate them into the appropriate format. A programmer at NIST was to lead the development of the data cataloging feature, which could be deeply developed before needing to be connected to the core. And hence was born DataView.
5.3: Design Highlights

When designing DataView, two primary objectives ruled our decisions: flexibility and ease of use, both for the casual user and the programmer. This plays out both in the front and back ends of the software. For example, the GUI is completely customizable through user profiles, selected at log-in and swappable at any point, allowing the user to tailor their experience (listed methods, displays, and so forth) to the nature of the data they are currently investigating. And for the programmer, adding new functionality is as easy as dropping a single file plug-in (coded following a template) into the correct directory. An automatic registration system recognizes the new material and automatically adds it to the code, adding necessary interface features, like menu structure, depending on the nature of the plug-in (be it viewer, analysis method, simulator, or so forth).

Below are a few feature highlights that showcase the scientifically relevant innovations in dataview.

5.3.1: History

As mentioned in the motivation, we firmly believe that, in a scientific setting, action logging is crucial during any interactions with data. The GUI driven analysis systems we investigated had no logging features. Command line or software driven analyses could, at least in theory, keep track of what was done. For example, a well designed Jupyter Notebook (a very nice tool for interactively performing data analysis in Python) consists both of the code performing the analysis and of annotations (user comments) about the procedure; a similar experience could be found in MatLab or Mathematica notebooks. Although these tools do a good job of showing a finished product, they often lose track of the process - the 3 steps forward, 2 steps back of data analysis – as the user often dives in and fine tunes parameters without recording the process or reasoning.

In DataView every step the user takes is logged. This history system is directly tied to the undo/redo system, so these steps are captured as well without the user having to give it a second thought (though annotation – the why – of analysis steps is strongly encouraged). Beyond this, history is also the base of a macro system, so that the user can repeat a series of analysis steps, either with the same or modified parameters, either on the same dataset or on one or more datasets of similar, though not necessarily identical, structure. This macro system gives the GUI user the power of a command-line user/programmer through an easy interface.

5.3.2: Data Generalization

Data in DataView is at once highly structured and very flexible. The structure is enforced to be able to think about the data scientifically. For example, each axis of the underlying numeric data must be labelled with a named “Dimension,” which keeps track of the values (with units) along that axis. But unlike the data structures in NISTView (and in other GUI-based SPM software), the dimensionality and meaning of the data is completely arbitrary. So, for example, a 2D dataset could be be a topography with “X” and “Y” dimensions but it could just as easily be a linecut with “X” and “Energy” dimensions.
Methods (processing and analysis plug-ins) are designed to respect this arbitrary structure. Each routine specifies requirements on the input data (e.g. the number and dimensionality of datasets it can work with); as long as the selected data meets those requirements, the method is available to the user.

More than this, however, because the data is labelled with dimensions, the user can easily think meaningfully about the process. In a 5 dimensional dataset, for example, the user could choose to do gaussian smoothing along the energy axis (1D) or spatially across the x-y plane (2D), simply by indicating how they currently want to think about their data (as specific collections of 1D or 2D datasets respectively) and then calling the same gaussian smoothing algorithm.

At the same time, the programmer of an analysis method that only works in 1D doesn’t need to worry about generalizing it to subsets of higher dimensional data. If the user is in the “thinking of this as a collection of 1D in energy spectra” mode, then low level code will essentially pass the data one spectrum at a time to the plug-in (note: optimization makes it not work exactly like this, but the principle of a coder only worrying about dimensionality at the level required by the routine stands).

5.3.3: Data Selectors and Viewers

This flexibility carries over to displaying data as well. Viewers, like methods, are plug-ins in DataView, so depending on the creativity of plug-in designers, a wide variety of displays could be available beyond simple 1D plots, 2D colormaps and 3D volumetric plots.

Each of these, however, could be used regardless of the full dimensionality of the underlying data. For example, a common STM data structure is the spectral survey, which is generally six dimensional (2 spatial, one energy, repeat and scan direction, and finally data channel, as multiple data, such as tip height, tunneling current, error signal, and so forth are often simultaneously recorded). These are often viewed as stacks of 2D images, as in Figure 5.1, with the ability to change which “layer” is being viewed (fix other dimensional values) via dropboxes at the viewer’s bottom. But the data could also be thought of as a collection of 1D spectra, such as in Figure 5-2, where the X,Y location of the spectrum is tied to the cursor in Figure 5-1.

This particular example is by no means unique. In fact, for the STM data collection software we use, Nanonis, this is the default view for spectral surveys. Unfortunately, as is often the case in STM software, this is also the only view for spectral surveys. In DataView, on the other hand, the user has complete control of which dimensions to view over and how to handle the other dimensions (such as selection via dropboxes or a cursor, or statistical collapse, like the calculation of a mean or extremum).
Figure 5-1: Example of an Image Viewer, showing a two dimensional image of conductance, with the X and Y coordinate axes being viewed over. The combo boxes at the bottom pick specific coordinates to select the layer in the multidimensional array. The cursor is connected to the plot viewer of Figure 5-2.

![Image Viewer Image]

Figure 5-2: Example of a Plot Viewer, showing a one dimensional plot of conductance, with the bias axis being viewed over. The data is updated when the cursor in Figure 5-1 is updated, selecting the data at the specific X and Y coordinate axes. In addition, when the cursor on this Plot is moved, the Bias axis in Figure 5-1 is updated. Other axes (direction and repeat #) are selected out via drop-boxes.

![Plot Viewer Image]
5.4: Summary

The above highlights should give a flavor of the motivating factors for the development of DataView – the ease and flexibility of both the casual user and the plug-in programmer to think scientifically about and deeply investigate their data. Of course, there are a multitude of other aspects of the code that could be discussed – Appendix E contains a more detailed view of many of these.

In the end, DataView is a primary product of my efforts as a physics graduate student. Though it is not a scientific result itself, I hope that it will be a helpful tool, used by many scientists to analyze their SPM and other scientific data. And, as it is designed to grow, I look forward to seeing, and perhaps participating in, its continued development.
Appendix A

Correlation Functions

Throughout this thesis I describe the relationship between variables in terms of their “correlation.” Depending on the nature of the variables and the relationship, I refer to one of several different correlation coefficients—Pearson’s, Spearman’s rank and the polychoric. Below I describe the calculation and rationale behind each.

A.1 Pearson’s correlation coefficient

Pearson’s is probably the most common of the coefficients, and measures the linear relationship between two variables, X and Y, measured in pairs \((x_i, y_i)\). It is mathematically defined as

\[
\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}
\]  

(A-1)

Here, \(\sigma_x, \sigma_y\) are the standard deviations of their respective variables, the square roots of the variances:

\[
\sigma^2_x = \langle (x - \mu_x)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2
\]  

(A-2)

which measures the spread of the \(N\) samples \(x_i\) around their mean \(\langle x \rangle \equiv N^{-1} \sum_{i=1}^{N} x_i \equiv \mu_x\). \(\sigma_{xy}\) is the covariance of the two variables

\[
\sigma_{xy} = \langle (x - \mu_x)(y - \mu_y) \rangle = \langle xy \rangle - \langle x \rangle \langle y \rangle
\]  

(A-3)

The coefficient, \(\rho_{xy}\), ranges from -1 to +1, indicating perfect negative to positive linear relationships and everything in between. Note that although there is no universal standard for “high” and “low” correlation, comparisons between correlations can reasonably be made for similar types of data, and there have been several rough guides for verbally describing the meaning of the absolute value of the Pearson correlation:

- .00-.19 “very weak”
- .20-.39 “weak”
- .40-.59 “moderate”
- .60-.79 “strong”
- .80-1.0 “very strong”
A.2 Spearman’s rank correlation coefficient

When it is the rank of variable values (1st, 2nd, 3rd...) that are cared about the Spearman coefficient is often reported. It is simply the Pearson coefficient for the rank of the values rather than the values themselves. This can be seen as relaxing the linear relationship demanded of the Pearson coefficient (any monotonic relationship between variables will yield perfect positive or negative correlation $r_s = \pm 1$). It is also commonly used when considering the relationship of continuous numeric and ordinal data, which frequently arises in working with categorical data (e.g. ‘good,’ ‘ok,’ ‘bad’) in machine learning.

A.3 Polychoric (latent) correlation coefficient

The final correlation coefficient I use in this thesis, the polychoric, or latent correlation, coefficient, is particularly useful when relating ordinal values. When the values being related come from a small set of categories the Pearson/Spearman coefficients can artificially suppress relationships. Calculation of the polychoric coefficient assumes that the measured values sample underlying normally distributed (“latent”) variables and attempts to essentially determine the Pearson correlation between those latent variables.

Details of the calculation of the polychoric coefficient are beyond the scope of this thesis (tools for computing the value are now available in most statistics packages). However, because the end result is essentially a Pearson correlation, the meaning of the numerical value can be considered similar to the rough guide presented at the end of section A.1.

A.4 Cross-Spectrum Analysis

In addition to correlating sets of values, we occasionally want to measure a correlation between two time dependent signals. One method, cross-spectrum analysis, is particularly useful if trying to understand the relationship of the signals in Fourier space.

The cross-spectrum $f_{xy}(\omega)$ is defined as the fourier transform of the cross-covariance function $\gamma_{xy}(h)$:

$$\gamma_{xy}(h) = \langle (x_{t+h} - \mu_x)(y_t - \mu_y) \rangle$$

(A-4)

$$f_{xy}(\omega) = \sum_{h=-\infty}^{\infty} \gamma_{xy}(h) e^{-2\pi i\omega h}$$

(A-5)

Here, $\mu_x$ is the mean of the signal $x_t$, $\mu_y$ is the mean of the signal $y_t$, and $\omega$ and $h$ a frequency and time shift respectively of interest. Although these can be used independently to understand the strength of a relationship, that is, how well one can predict an output series $y_t$ from an
input series $x_t$, the more typical way of analyzing that strength is through the \textit{_squared coherence} function, defined as:

$$
\rho_{y,x}^2(\omega) = \frac{|f_{xy}(\omega)|^2}{f_{xx}(\omega)f_{yy}(\omega)}
$$

(A-6)

This is completely analogous to the square of Pearson correlation, described in section A.1. It is particularly useful though, as it can be used to calculate the mean squared error of an equivalent linear lagged regression model – that is, to tell us how well we will be able to estimate $y$ from $x$. A lagged regression model is similar in form to the convolutional transfer function model used in ANITA (Chapter 3).

For a lagged regression model, which tries to estimate a time series $y_t$ from $x_t$ using coefficients $\beta$:

$$
y_{t,\text{Estimate}} = \sum_{r=0}^{\infty} \beta_r x_{t-r}
$$

(A-7)

the mean squared error is:

$$
MSE = \langle y_t - \sum_{r=0}^{\infty} \beta_r x_{t-r} \rangle
$$

(A-8)

which in terms of the squared coherence, $\rho_{y,x}^2$, and the spectral density of $y$, $f_{yy}(\omega)$, is:

$$
MSE = \int f_{yy}(\omega)[1 - \rho_{y,x}^2(\omega)] d\omega
$$

(A-9)

Looking at this equation, the mean squared error between prediction and real values approach zero when the squared coherence is equal to 1 at all frequencies.
Appendix B

Size of Scanning Tunneling Microscopy Data

This dissertation discusses the analysis of data from scanning tunneling microscopy. STM data comes in many different shapes and sizes depending on the nature of the experiment. In this appendix I will describe some typical types of experimental measurements made on our instrument, and summarize the dimensionality and size of data associated with these measurements.

B.1 Data Dimensions

Traditionally, STM data was considered either one dimensional (“spectroscopy”) or two dimensional (“topography”). In the late-1990s the idea of a recording a “spectral survey” (a series of spectra on a 2D grid) arose, leading to 3D data.

Modern STM experiments can record a variety of types of data as a function of a number of different variables. So in addition to the $X$ and $Y$ dimensions of the topographic grid and the Energy (or Bias dimension) which is ramped, for example, for differential conductance spectroscopy, our spectral surveys typically have three other dimensions. First, due to the nature of how STM data is scanned, there can be differences in data if the data was scanned from left to right or right to left in the case of a topography, or ramping up or ramping down in voltage in the case of a spectroscopy – this leads to a Direction dimension. Second, the measurements are often repeated several times. And third, it often happens that more than one kind of signal is recorded during a survey. For example, conductance (which, as the output of a lock-in, has both in phase and out of phase signals), current, and topography may all be measured simultaneously. This leads to a Data Channel dimension. The number of data channels is very dependent on the acquisition system. Nanonis, which we use, tends to load up on channels, to make sure that we have everything we could possibly need.

B.2 Data Sizes

Compared to many physics experiments, especially high energy experiments, the amount of data recorded in a single STM experiment, and the data rate, are quite small. However, because many of our data analysis techniques require creating data structures in memory that are many times the size of the original data being analyzed (for example, for convolutional neural networks – as in Chapter 4), it is worthwhile to think about the sizes of typical STM measurement datasets. It is also worth noting that although the acquired data is typically 16 bit
(=2 byte), during analysis we often work in double precision floating point (64 bits = 8 bytes). Table B-1 shows typical dimensions and array sizes of common STM datasets (ignoring the header size, which is typically negligible).

Table B-1: Common STM Dataset Sizes (double precision floating-point). The size of various dimensions for typical datasets. The acquisition (in file) and analysis (in memory during calculations) sizes differ by a factor of 4, as described in the text.

<table>
<thead>
<tr>
<th>Description</th>
<th>X</th>
<th>Y</th>
<th>Bias</th>
<th>Direction</th>
<th>Repeat</th>
<th>Channel</th>
<th>Acq Size</th>
<th>Anal Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Topography</td>
<td>256</td>
<td>256</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>500 kB</td>
<td>2 MB</td>
</tr>
<tr>
<td>Typical Topography</td>
<td>512</td>
<td>512</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>2 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Large Topography</td>
<td>1024</td>
<td>1024</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>4</td>
<td>16 MB</td>
<td>67 MB</td>
</tr>
<tr>
<td>Single Spectrum</td>
<td>-</td>
<td>-</td>
<td>101</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>24 kB</td>
<td>97 kB</td>
</tr>
<tr>
<td>Good Spectrum</td>
<td>-</td>
<td>-</td>
<td>201</td>
<td>2</td>
<td>200</td>
<td>6</td>
<td>1 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td>Line Cut</td>
<td>512</td>
<td>-</td>
<td>101</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>12 MB</td>
<td>50 MB</td>
</tr>
<tr>
<td>Small Survey</td>
<td>256</td>
<td>256</td>
<td>41</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>75 MB</td>
<td>300 MB</td>
</tr>
<tr>
<td>Typical Survey</td>
<td>512</td>
<td>512</td>
<td>101</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>1.5 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>Large Survey</td>
<td>1024</td>
<td>1024</td>
<td>101</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>30 GB</td>
<td>119 GB</td>
</tr>
</tbody>
</table>
Appendix C

Programming a Convolutional Neural Network in Python

The following code shows example code of training a convolutional neural network model coded using the Keras package in Python.

1. # model creation
2. model = Sequential()
3. model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                  activation='relu',
                  input_shape=input_shape))
4. model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
5. model.add(Conv2D(64, (5, 5), activation='relu'))
6. model.add(MaxPooling2D(pool_size=(2, 2)))
7. model.add(Flatten())
8. model.add(Dense(1000, activation='relu'))
9. model.add(Dense(num_classes, activation='softmax'))

These lines in the code explain the development of the architecture of the model – while the researcher doesn’t need to know the linear algebra behind each layer of the model, they do need to know the format of the types of layers and the parameters involved in the layers. Here, the researcher has created a simple CNN composed of two blocks of convolutional and pooling layers, followed by a fully connected layer, and an output layer with softmax activation. It’s important to note that all of these layers are chosen by the researcher.

The next step is telling the framework the type of loss function to use for the model and which optimizer the researcher plans to use. In Keras, this is done with a single command:

12. # compile model
13. model.compile(loss=keras.losses.categorical_crossentropy,
                 optimizer=keras.optimizers.SGD(lr=0.01),
                 metrics=['accuracy'])

Keras supplies many different types of loss functions. In this case, categorical cross entropy is the standard loss function for multiclass classification. The optimizer chosen here is a form of stochastic gradient descent with a specifically chosen learning rate.

The next step is training the model, also run by a single command in Keras:

16. # train model
17. model.fit(x_train, y_train, batch_size=batch_size,
              epochs=epochs, verbose=1,
              validation_data=(x_val, y_val),
              callbacks=[history])
In advance, the researcher has already preprocessed the training data and validation data (x_train and x_test) as well as their corresponding labels (y_train and y_val). This is one particular example of how to train the data, in which the researcher passes all of the data at once into the algorithm at once. We choose how the size of batching in our data – during a single “epoch”, an optimizer only looks at a chunk of a data at a time but eventually pass through the whole dataset. We also set the number of epochs, which determines how many passes looking through all the data the model will train on. Keras supplies many different types of loss functions. In this case, categorical cross entropy is the standard loss function for multiclass classification. The optimizer chosen here is a form of stochastic gradient descent with a specifically chosen learning rate.

Finally, the model needs to be evaluated on the validation data:

21. # evaluate model
22. score = model.evaluate(x_val, y_val, verbose=0)
23. print('Validation loss:', score[0])
24. print('Validation accuracy:', score[1])
Appendix D

Review of Scanning Probe Microscopy Analysis Packages

There are a number of commercial and freely available products to analyze scanning probe microscopy data. This appendix will list the most popular SPM analysis packages, and describe their strengths and shortcomings. The analysis packages come in two flavors: Packages focused on image processing/data analysis alone, and packages that offer both image processing and data acquisition modes.

Table D-1: Scanning Probe Microscopy Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>GUI</th>
<th>Custom Plug-Ins</th>
<th>Data Acquisition</th>
<th>Grid Spectroscopy</th>
<th>Multidimensional</th>
<th>Open-Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gwyddion</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>WSXM</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>GSXM</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SPIP</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Pycroscopy</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DataView</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

The following is a description of the different categories of this table:

- **GUI**: This package is based off of a GUI, rather than a command-line system or package for a programming language.

- **Custom Plug-Ins**: Does the package allow the creation of custom plug-ins by the user.

- **Data Acquisition**: Whether this package has a module for data acquisition in addition to image processing.

- **Grid Spectroscopy**: Whether this package handles grid spectroscopy: layered 2D views of spectroscopic maps.

- **Multidimensional**: Whether the format of the datasets involve are flexible and N-dimensional rather than rigid dimensional formats (“line”, “image”, “grid” objects)

- **Open-Source**: Whether the package is open source.
The following is a description of the packages described in the table above:

**Gwyddion**: Gwyddion is a modular program for SPM (scanning probe microscopy) data visualization and analysis. Primarily it is intended for the analysis of height fields obtained by scanning probe microscopy techniques (AFM, MFM, STM, SNOM/NSOM) and it supports a lot of SPM data formats. However, it can be used for general height field and (greyscale) image processing, for instance for the analysis of profilometry data or thickness maps from imaging spectrophotometry. Available for Linux, Windows and MAC OS. Frequently updated.

**SPIP**: An advanced software package for processing and analyzing microscopic images at nanoscale and microscale. It has been developed as a proprietary software by Image Metrology and is unique in the microscopy and microscale research market. Has a purchase price, but a time-limited demonstration version is available. Frequently updated.

**WSxM**: Freely available software that supports many SPM file formats; and has many analysis tools. I personally like a lot the 3D rendering results from WSxM. It was originally developed by an AFM manufacturer for use with their instrument, but is now completely independent and supports very many other file formats. Unlike many third party programs, has support for force curves as well. Frequently updated.

**GSXM**: The GXSM software is a powerful graphical interface for any kind of 2D and up to 4D (timed and multilayered 2D mode) data acquisition methods, but especially designed for SPM and SPA-LEED, which are used in surface science. It includes methods for 2D data (of various types: byte, short, long, double) visualization and manipulation. It can be used for STM, AFM, SNOM, SPA-LEED, but is by far not limited to those! Especially in standalone mode it can perform many SPM typical image manipulations and analysis tasks. Latest additions enables full support of handling and on-the-fly viewing image sequences and arbitrary profiling in 4 dimensions.

**Pycroscopy**: Pycroscopy is a python package for image processing and scientific analysis of imaging modalities such as multi-frequency scanning probe microscopy, scanning tunneling spectroscopy, x-ray diffraction microscopy, and transmission electron microscopy. Pycroscopy uses a data-centric model wherein the raw data collected from the microscope, results from analysis and processing routines are all written to standardized hierarchical data format (HDF5) files for traceability, reproducibility, and provenance.
Appendix E:

DataView Programmer’s Guide

E.1: Delving into DataView

Before a programmer can implement modules, it is important to understand the Python packages integral to their functionality. DataView heavily uses object-oriented programming within Python, and it is important for a programmer to understand concepts such as classes, instances, methods, inheritance, attributes, and properties. DataView is intended to be used with the Anaconda distribution, which contains most of the prerequisite packages. The six most important packages are NumPy, SciPy, H5Py, Pint, Matplotlib, and PyQt.

E.1.1: How to install DataView

DataView is hosted on GitHub. It is developed in Python 3.6 using Anaconda and the JetBrains PyCharm IDE. The following are installation directions for Windows machines to get up and running with the full package.


- If you are using windows, make sure that your Anaconda folder (typically Anaconda3) and the Anaconda\Scripts folder are added to your PATH as an environmental variable.

- Create a new Python Environment in Anaconda called dataview. You can do this by opening Anaconda Navigator, clicking "Environment" on the left-hand tabs, and clicking "Create" to create a new environment. Name this dataview and make sure that is a Python 3.6 environment. Then, go to a command prompt (any directory is fine - as long as Anaconda was properly installed it should be in your path), type activate dataview (if windows) or source activate dataview (if mac/linux) to activate the dataview Anaconda environment, and type conda install packageName or conda update packageName to install the following packages:
- configobj (Python module for easy reading and writing of config files134)
- numpy (fundamental package for scientific computing with Python75)
- scipy (a Python-based ecosystem of open-source software for mathematics, science, and engineering135)
- matplotlib (2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments136)
- h5py (Pythonic interface to the HDF5 binary data format117)
- scikit-learn (a package for machine learning137)
- scikit-image (a collection of algorithms for image processing138)
- pillow (a "friendly PIL fork"139 (Python Imaging Library))
- pep8 (a python style guide checker140)
- psutil (cross-platform library141 for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python)
- pylint (code analyzer142-- works with pep8)
- pyqt (Python bindings for the Qt cross platform GUI toolkit143, we are using version 5)

We will undoubtedly add further packages. Some packages should be installed with pip instead using pip install packageName:
- pint (Package for units144)

  • For the IDE, register as an educator or student (assuming you are) at JetBrains and download PyCharm developer for free.
  
  • When running PyCharm, make sure that your python interpreter is set to the python.exe in your dataview environment, typically in Anaconda3/envs/dataview/python.exe.

Finally, you should pull the latest master branch of DataView form GitHub using your favorite method. The GitHub location is https://github.com/ericwhudson/DataView/. If you want a command line version of git, you can download git for windows145. If you want a plug and play version of git, you can download GitHub Desktop146.
Once the installation of Anaconda and all the relevant packages is complete, you can load DataView through one of a few options.

- Run PyCharm. Open the DataView package within PyCharm, and run DataView.py.
- Run `DataView.bat`, (if you are using Windows) assuming you are running DataView from the dataview environment. If you are not, edit this file to account for the correct anaconda environment.
- You can run the script manually. Open up a command line with the dataview anaconda environment, opened with `activate dataview` (if windows) or `source activate dataview` (if mac/linux) and type `python DataView.py` when you are in the directory which contains this file.

E.1.2: Anaconda

DataView is built on the Anaconda distribution of python, a popular open-source data science ecosystem. It is intended to be used for large-scale data processing, predictive analytics, and scientific computing, and is an easy way to install most of the necessary packages for DataView without having to download or compile them separately. Using the package management system `conda` included with Anaconda, you can install, run, and update packages and their dependencies. Python’s `pip` package may be used for packages with python-only dependence, such as `Pint`, but packages which have library dependencies outside of Python, such as `NumPy`, `H5Py`, and `PyQt`, should be installed using `conda`.

E.1.3: NumPy

`NumPy` is a python package which adds support for large, multidimensional arrays. It also has a large collection of high-level mathematical functions to act on these arrays. Python is an interpreted language, and mathematical algorithms typically run much slower compared to compiled languages like C or Java. An interpreted language execute instructions directly, instead of having to compile a program into machine-readable instructions – an interpreted language is typically faster to start up from raw code, but slower when performing calculations. The purpose of NumPy is to create operators which act efficiently on arrays, allowing one to rewrite inner loops quickly. It is functionally comparable to MATLAB, a closed-source numerical computing environment, as both are interpreted. It allows for fast computation as long as most operations
work on arrays instead of scalars, and is integrated into Python. The core functionality of NumPy is the 
ndarray, an n-dimensional array, which are strided views on memory. In contrast to
Python’s built-in list structure, which is a dynamic array, these arrays are homogeneously typed,
as all elements of a single array are built into a single type. DataView’s in-memory data objects
are ultimately wrapped in NumPy’s array object.

E.1.4: SciPy

The SciPy library is a python package used for scientific computing. It contains
functions for optimization, linear algebra, integration, interpolation, special functions, FFT,
signal and image processing, ODE solvers and other tasks common in science and engineering. It
builds off NumPy’s array object, and is part of the SciPy stack, a collection of open source
software for scientific computing. SciPy’s functions are typically used in DataView’s methods to
manipulate and create new datasets.

E.1.5: H5Py

H5Py is a python package which implements a pythronic interface to the HDF5 binary format, a
file format designed to store a large amount of data. Using H5Py, you can easily manipulate the
data from NumPy, such as slicing into multi-terabyte datasets stored in disk as if they were real
NumPy arrays. It uses straightforward NumPy and Python metaphors to make the integration of
data on disk seamless with python. H5Py is used for both the in-file data system as well as its
default data storage format.

E.1.6: Pint

Pint is a python package which defines, operates, and manipulates physical quantities, which are
the product of a numerical value and a unit of measurement. Using Pint, you can allow for
arithmetic operations between physical quantities, such as adding two lengths together, or
extrapolating force from mass and acceleration. You can also convert to and from different
units, such as converting to inches from centimeters. Pint is used in DataView in its unit system
implemented in its Dimension and Converter objects. The most important class in Pint used in
DataView is its Unit Registry, an object within which units are defined and handled. Unlike the
other packages mentioned here, Pint is not located in Anaconda and must be installed
separately using pip. Pip is a command-line tool for installing python packages, focused on
python library dependencies only – packages which need to handle library dependencies outside
of the Python packages (such as C dependencies in NumPy) are better handled using conda.
E.1.7: Matplotlib

*Matplotlib* is a two dimensional plotting library for python, which produces publication quality figures in a variety of formats and interactive environments. Matplotlib can generate structures such as plots, histograms, power spectra, bar charts, scatterplots, and more. While Matplotlib is typically used in a few lines of code in a MATLAB-like interface in interactive environments such as the *Jupyter* notebook, DataView uses its full control of line styles, font properties, axes properties, and more using its object oriented interface and sets of functions to create fully descriptive plots encased in its graphical user interface. Matplotlib typically works on Numpy arrays, the underlying in-memory structure for DataView’s data objects. Typically, DataView works on subsets of its large, multidimensional data objects to view selections of its data in matplotlib plots. Matplotlib is one component of DataView’s Viewer system, used for a significant number of the widgets encased in the Viewers. The most important class in Matplotlib to understand is the Axes class, which contains most of the figure elements for plots and the coordinate system.

E.1.8: PyQt

*PyQt* is a python package which implements python bindings to the Qt GUI toolkit, a cross-platform application framework used to develop multi-platform applications and graphical user interfaces. DataView uses PyQt version 5 (PyQt5), the version contained in the latest version of Anaconda. PyQt is used profusely in DataView’s Viewer system.

One key feature of PyQt is widgets. The widget is the atom of the user interface, which receives events like mouse and keyboard clicks from the window system and paints a representation into the screen. Widgets include windows, buttons, and layouts, all of which are necessary for a full Python Viewer.

Another key feature of PyQt is its use of signals and slots between objects, which encourages the use of development of reusable components. A signal is emitted when something of interest happens, while a slot is a python callable function. If a signal is connected to a slot, then the slot is called when the signal is emitted. If a signal is not connected, then nothing happens. The code that emits the signal does not know or care if the signal is being used – it is independent of the slots connected to the signal.

E.2: Subpackages

DataView is organized into a number of subpackages. It contains a registration process to adaptively modify and integrate add-ins into the GUI-based program. With the exception of the low-level data code, the structure of most of the components is similar. Each component has a
Base class, specific to the component, in addition to its own metaclass, derived from python’s `ABCMeta` class to allow abstract routines and enforced routine implementation, in addition to a container of all classes that implement the base. Each modular component can be created by subclassing the base class of the component, using Python’s object inheritance. The following are a list of the different subpackages and folders within DataView in alphabetical order.

**E.2.1: data**

The `data` subpackage contains the backend low level data structure that the rest of the program is built from. The main data structures include classes grouped into dimension classes, converter classes, data classes, and locator classes. Dimension classes include Dimensions and the containers which store them. Dimensions connect indices of a data array with values, such as coordinates, which contain units. As we often need to convert from one number to another – such from indices to values or different unit systems, Converter classes have been designed to handle potentially complex conversion processes between numbers. The data are stored in the Data packages, which include the data objects and containers of data objects. This includes a wrapper (a class which encapsulates the functionality of another class) around the raw array so we can handle in-memory and in-file data in similar ways. It also includes the `DataSet`, the basic data unit, containing the numerical data itself and a set of dimensions. Locator classes are objects which store selected indices and dimensions of the `Dimension` class, which can dynamically change to manipulate a subset of data. Data can be subset with `DataSelectors`, which are composed of a chain of tasks, a modular set of classes stored and registered in the `dstasks` folder. As methods need to iterate on slices of data objects, they are iterated with the `DataIterator` class, a high-level iterator class which helps the programmer to flexibly manipulate the multidimensional datasets. The `DataObjectChooser` class is used to control the interface between input data objects and DataIterators.

**E.2.2: database**

The `database` subpackage contains the structure of the program which connects to DataView’s database. Scanning probe microscopes produce vast quantities of data, and it is important to categorize them. Typically, a researcher would have to use a log book which records the date, type of sample, and other relevant information – when scouring for this information, the researcher would have to go through this log book, and find the file they are looking for. This is not a practical solution. The database has the ability to scrape metadata from files, such as thermometry, magnetic field applied on a sample, information about the sample and tip. It throws this metadata into a single place, linked to the file of the dataset. The user will have the ability to search datasets based off this information. Being a structured dataset, it will also be
possible to use this database for use in supervised or unsupervised machine learning, combining the metadata with the multidimensional dataset for predictive purposes.

E.2.3: filehandlers

The filehandlers subpackage contains the FileHandler modules, which are modules that handle loading and writing from data files. FileHandlers are composed of a class with a number of class methods, which include functionality including loading from a file, saving to a file, file configurations, and how to display the data by default. They could also have additional helper methods on a module by module basis which aid in the file handling process.

E.2.4: fitfunctions

The fitfunctions subpackage contains information about functions used for curve fitting for spectroscopy. It is useful for fitting one dimensional spectra. The important aspect of these functions is extracting the parameters of the curve. For example, a researcher may want to fit a superconductor’s density of states spectra to dynes formula to extract the superconducting gap and broadening factor across the entire image.

E.2.5: main

The main subpackage contains classes important for gluing DataView into a coherent whole. Most importantly, the main entry point into the program is located here, as well as the main GUI window, main menu and menu system. The history system contains the underlying elements for the undo/redo functionality of the program, and includes the History class, which stores a list of actions which can branch off into multiple lists – a tree of the history. The subpackage contains the DVPreferences class, which handles interaction with preference files, and for the information which is read by default on the start of the program. Finally the package contains the login box for the user to select the name and profile at the start of the program.

E.2.6: methods

The methods subpackage contains routines which process, analyze, or display data. Methods use Dataview’s modular registration system. Most methods act on DataIterators, which contain all the information needed to manipulate data. Analyze methods interpret and create new data without modifying the data, and will typically create a new Viewer to display the new data. In contrast, Process methods directly modify the data under consideration; as data is modified, all
process methods are undoable. Display methods are designed to modify or create new viewers, and act on viewers, not iterators. A Data Object Chooser dialog may pop up for the user if the method requires more than one data object to pass through.

E.2.7: preferences

The preferences folder is not a subpackage per se, but contains all the initialization files necessary to run DataView. DVPref.ini contains information about all users registered into DataView, including the names of their profiles and the location of their local preference folder. Other files in this folder are the *.ini preference files themselves, following a [user]_[profile].ini named structure. The initialization files themselves contain information about the different modular components of DataView and are customizable. It is necessary to make sure that the preferences are correct on the local computer before DataView can run.

E.2.8: simulators

The simulators subpackage contains all classes related to constructing simulated data that can be analyzed in DataView. If a researcher is making a model of what they think is going on in the system, they need to be able to generate simulated data and see how the model data is similar to the experimental data being taken. In addition, when a programmer is writing routines to analyze data, it is sometimes easier if they have a model. A routine meant to extract parameters can be tested by first creating a simulation which is based off these parameters, and seeing if they can get the parameters out of the model. Examples of methods which would be useful to tie to a simulator include thermal broadening and noise broadening of one dimensional spectroscopy.

E.2.8: utilities

The utilities subpackage contains various python functions and classes which are used as accessories in the other subpackages. For example, there is a function to determine if data is numeric, a function which implements an arbitrary polynomial, and functions useful as accessories in Methods, such as a function which creates a DataSelector based off an old one. Unlike other subpackages, functions in utilities tend to not be modular, as they are simply imported into the relevant modules as necessary.

E.2.9: viewers
The viewers subpackage contains Viewers, LocatorWidgets, and the ViewGroup. Viewers are objects which display the data indicated by a DataSelector. Viewers could be based off of Qt Widgets, Matplotlib widgets, or something else as long as it has the same basic structure as the object the viewers inherit from, ViewerBase. Viewers are stored within a ViewGroup, a window which serves as a GUI container to hold multiple viewers at once. LocatorWidgets are wrapper on GUI Widgets, which are stored in Viewers and are the connection between the display and selection of data within Locators. Both Viewers and LocatorWidgets use DataView’s modular registration system.

E.3: Data Flow

Before we delve into the details of the program, it is important to know the structure of the flow of data in the program. Figure E-1 shows the data flow – the data starts from a DataSet, a data structure which contains information about the numerics and dimensions of the data. It is subset by a DataSelector, an object which slices and reshapes DataSets to create subsets of data. The DataSelector is visualized by a Viewer, which contains the graphical user interface elements of a window of the program. An element of a Viewer is a LocatorWidget, such as a combo box or cursor, which when signaled by something like a mouse click, triggers a Locator to update a DataSelector. A DataSelector may be acted on by a DataIterator, an object which contains information about how to iterate the data. The DataIterator in turn is iterated over by a Method, which modifies or creates new data in the form of a DataSet.
E.4: Data Classes

DataView is based off of Data. This section thoroughly describes the purpose and different ways to use the different Data classes. The attributes, parameters, methods, and properties of these classes are described. As a reminder:

- An attribute is a variable attached to a class
- A parameter is a variable used to initialize a class
- Methods are functions attached to a class
- Properties allow access to and setting of variables attached to a class
The above figure shows how the different data structures in the software are connected to each other. Some classes inherit others – a LocatorDimension is a subclass of a Dimension, and inherits all attributes and methods of a Dimension. Some classes are best thought of as containers of others, or contain containers – a DimSet is a container of Dimensions; meanwhile, Links work by connecting two dimensions together and need to have a container of dimensions. Finally, a data structure might need information about another data structure – for example, a DataSet consists of a DataBlock and a DataSet. Yellow corresponds to dimension classes, blue to dimension classes, orange to numerical data classes, purple to data collection classes, red to dataselector classes, and grey to miscellaneous objects acting on data classes.

**E.4.1: Dimension Classes**

The following classes are most of the classes related to the dimension aspect of data. The dimension classes related to Locators are in a later section.
**E.4.1.1: Dimension**

A **Dimension** is a class which describes the real world properties of an axis of a DataSet. It is different from an “axis” – which is the numerical axis of the corresponding array, separate from the Dimension associated with it. Rather than simply being a numerical index into a DataSet’s array, a Dimension knows about the real world meaning of the index, allowing both input and output in real world units. It connects indices with values. For example, if an axis had 1000 points with 5 mV spacing (from 0 to 5 V), you could either ask for index 200 or index 1 V, and when finding the data in the array, the location could come back as a numerical index or as a value index. A Dimension is primarily characterized by its name, size, and unit, although there are other important elements as well related to the formatting of the Dimension and conversion process. Values are calculated using a Converter, which converts between indices and real world units. A Dimension does not store the axis it is held in. This information is stored elsewhere, because an identical Dimension can be used in different selections of data. Slicing over a Dimension always returns a new dimension, but with a potentially different size and values. Iterating over a dimension yields the potential values of the dimension. Useful methods include returning a dimension’s numerical value, with units or just the numerical value, a shortcut linear spacing method, and a way to get the index for a given real world value.

**E.4.1.2: DTDimension**

A **DTDimension** (short for “Datatype Dimension”) is a special kind of Dimension (subclassed from Dimension) that knows about the units of the data in the DataSet. Every DimSet has one and only one DTDimension. This Dimension may or may not be an “axis” in the conventional sense, although it’s not stored in the Dimension itself. For example, if you have a 2D data set of current measurements vs. position, the axes would be X & Y, but the DTDimension would tell about the measurement itself (e.g. that the values are in Amps). In this case the DTDimension isn't a traditional axis (it would have axis=-1) and wouldn't appear in the shape of the DimSet. If there are multiple Datatypes in the data however, for example if you simultaneously recorded Z and I vs X,Y then the DTDimension would be a real axis (probably axis=2, with X & Y being 0 & 1 respectively). A DTDimension has analogous attributes to a Dimension, but because each datatype is different, it stores its names, units, and formats as lists instead of individual attributes. It also stores a “dtype”, a NumPy object which describes the format of the underlying Numpy array associated with the dataset that the DTDimension describes\(^1\). Slicing a DTDimension returns a DTDimension, typically of a single channel. Iterating over a DTDimension iterates over the names of the units.
E.4.1.3: DimSet

A DimSet (short for “Dimension Set”) is a collection of Dimensions. A DimSet collects a set of uniquely named Dimensions that completely define the axes of a DataSet. In order to be considered complete, it must contain one and only one DTDimension, although that dimension’s axis need not explicitly appear in the associated DataBlock. Dimension sets are named, ordered sequences. You can index a DimSet by integer (the axis corresponding to a Dimension), by name (the name of the Dimension), or a list of keys to generate a list of a subset of the dimensions stored. Iterating over the DimSet yields the stored axis dimensions. A DimSet also stores a container of LocatorDimensions, called a LocatorDimSet – both of these objects are described in section E.1.3. It is possible to obtain the axis or name of a Dimension stored in the DimSet, useful when this information isn't generically known, as is the case when working with Viewers. There are also helper methods for slicing and manipulating dimension sets in analogous ways to how arrays are manipulated. Another important method checks to see whether the DTDimension is an axis dimension or not, useful in modules such as viewers or methods, as DTDimensions may need to be handled differently from other dimensions.

E.4.1.4: Link

A Link identifies two dimensions as “the same” and gives them a way to convert from one dimension to another. For example, in pixel space, either all the pixels are the same, or there must be a way to transform one set of pixels to another. One Dimension may be half as large as another and every pixel in that dimension corresponds to every other pixel in the other dimension. The Dimensions’ real world units would have the same start and endpoints in real world units, are spread out in similar fashions and most likely have the same name, but could, for example, have a different size. They are typically created in the Data Object Chooser dialog when a method treats some of multiple input dimensions the same. (This is not yet implemented.)

E.4.1.5: Converter

We will often need to convert from one number to another and back again. For example, in Dimensions we need to convert between pixels and real world units. If the conversion is just standard unit conversion we can use Pint, but for anything more complex we use Converters. At their heart they simply store two functions, one for the forward conversion and another for the backward. Converters have a simple module system, with their own metaclass and registration process; unlike other modules they are stored in a single script,
data/converter.py. The converter base class is simply called “Converter”. Converters contain information for transforming between two sets of values. There are a few abstract methods that must be overridden by derived classes. The units are stored in a 2-tuple, the first element corresponding to be the unit converting from, and the second element corresponding to the unit converting to. A converter also contains specific parameters for how to do the conversion. Methods include ways to convert the index to a value and vice-versa, as well as a way to check that the parameters are of the right format. Common converters include Linear (a linear conversion between index and value), Values (stores a list of values and looks up this list for conversion), and Null (index and value are the same, typically used for unitless dimensions).

E.4.2: DataSet Classes

The following classes are all the classes relating to the numerical portions of data, starting from the very bottom, wrappers on arrays, to the very top, containers of high-level data storage classes.

E.4.2.1: DataBlock

The DataBlock is a way of referencing an array of data, with a wide variety of functions for indexing and slicing. The actual storage of data is transparent to the user. It is a class wrapped around a numerical array, such as a NumPy array\textsuperscript{154}, NumPy masked array\textsuperscript{155} or HDF5 “dataset”\textsuperscript{156}. This low-level class has been constructed so that you can apply functions on both in-memory NumPy arrays (either an n-dimensional array or a masked array) and in-file HDF5 files, and has the potential to be expanded further for other array formats. As H5Py’s HDF5 handler system is adapted to be analogous to NumPy, this makes DataView’s indistinguishability between the objects easy. However, the underlying HDF5 objects are more difficult to perform computations over compared to NumPy. A good number of the useful attributes stored in the array can be accessed here. There is a special method to apply a function on the array which returns a DataBlock, although typically the output array of this method would be a NumPy array. Some additional methods include cloning, checking whether the DataBlock is a view of another DataBlock analogously to NumPy’s view\textsuperscript{157} as well as helper methods for manipulating the underlying array. Additional methods for the DataBlock will be ways to wrap important array manipulation methods to seamlessly integrate both NumPy arrays and HDF5 data manipulation.
E.4.2.2: DataSet

The *DataSet* is the fundamental data unit in DataView. It is essentially a labelled array, containing a *DataBlock* and a *DimSet*, as well as an arbitrary header and a History. All array checking is handled by methods analogous to NumPy’s methods, so you can determine a *DataSet*’s shape, number of dimensions, and NumPy dtype. A *DataSet* may contain a container of Bound Selectors, a number of DataSelectors attached to the *DataSet*, with a limited number of tasks that can be applied to them for an easy interface typically used to quickly create them in file handlers. A *DataSelector*, as explained in section E.1.4.1, is an object which slices and dices a *DataSet* to create subset of data. A *DataSet*’s *DataBlock* and *DimSet* must correspond to each other – the shape of the *DataBlock* must match that of the *DimSet*, for example. A *DataSet* can be sliced over like its lower level components, creating a *DataSet* with a corresponding sliced *DataBlock* and *DimSet*. A *DataSet* has a method which determines if the data is large, the value of which is an amount of memory set by a user’s preference, which determines the switch from in-memory to in-file *DataBlock* usage.

E.4.2.3: DVCollection

A *DVCollection* is an assembly of related DataView objects, and is the main container of objects in the program. They are stored in a hierarchy – one *DVCollection* can be stored in another. Objects inside a *DVCollection* can be called by index key or name. Typical objects stored in a *DVCollection* include *DataSets*, *DataSelectors*, *Viewers*, and other *DVCollections*, but it can potentially include any type of object that is related to each other. Iterating over a *DVCollection* will drill down the hierarchy, obtaining all items that are not *DVCollections* themselves. On the front end they are useful as viewable “folders” of DataView objects.

E.4.3: Locator Classes

The locator classes are objects which mediate indexing of one or more values from a list of possible values. Locators are built on the generic Locator base class, described below. This section also contains information on the dimension classes associated with Locators.

E.4.3.1: Locator

The Locator is the base class for all Locator objects, including Pickers, PickerArrays, and PickerClusters. All Locators are built similarly like the Locator. The Locator is built on the “index”
property, a tuple of index objects, which could be an integer or an indices array. They are constructed from PyQt QObjects to engage PyQt’s signal and slot system. Specifically, it emit a valueChanged signal when its underlying “index” property is changed, as well as a slot that can be connected to any other signal. A Locator holds a list of dimensions which correspond to the index of the Locator and is typically called by methods that handle information about Locators. Locators also have helper methods for its signal system, such as connecting, and handling.

E.4.3.2: Picker

The Picker is the simplest Locator, and handles the selection of a single element within one or more Dimensions, and handles communication of that element’s selection between objects that care, such as LocatorWidgets and DataSelectors. They typically correspond to combo box or cursor widgets. For example, a DataSelector might have a picker over the X & Y coordinates of a dataset. An image viewer could set that Picker value via a cursor LocatorWidget, and a plot viewer could use that value to look at a curve from that location. The Picker only returns a single point, rather than a collection of points. All indices in the index property are passed in tuples, including one dimensional pickers.

E.4.3.3: PickerArray

A PickerArray handles multiple picks at a time, and are typically handled by DataView’s Region of Interest (ROI) widgets. The “index” property acts differently here, storing its values as a tuple of NumPy index arrays\(^{158}\) that can be used to slice to reduce the dimensions of the DimSet. The index arrays may be defined through many different methods, although the most straightforward case is that of a line cut, which reduces two or more coordinate dimensions to a single dimension across a line. Similarly, multiple line cuts can be done to create multiple dimensions like collapsing multiple dimensions. The PickerArray has the same signal and slot system as a Picker. Similarly to pickers, all index arrays are passed as a tuple, including one dimensional PickerArrays. There is a method which implements line cut selection from a start and end point of indices.

E.4.3.4: PickerCluster

PickerCluster creates a set of PickerArrays. It is used to select multiple sets of points at once, such as identifying sets of different objects in an image. A PickerCluster is constructed from lists of index arrays, a level of complexity higher than PickerArrays. A PickerCluster can have different
lengths. For example, the PickerCluster may have N objects, but the number of pixels forming each object can be different. A task asking on a PickerCluster will typically be followed by a function which collapses a ragged axis, such as some statistics like averaging. This “ragged axis” is implemented on the low level as a NumPy Masked Array wrapped within a DataBlock, with the data points not implemented on a row of the axis as masked points. The use of a masked array allows most array functions to act only on the non-masked points. A PickerCluster is typically applied by an ROI widget.

**E.4.3.5: LocatorDimension**

A LocatorDimension is a special dimension (subclassed from Dimension) which is either defined or collapsed by a locator (a picker, pickerArray or pickerCluster). In some cases these Dimensions don't yet exist in the associated datablock (they will be created by a pickerArray, like a line cut for example). In other cases they are Dimensions that need to go away. On initialization it must be called with the original dimension, which is then converted into a LocatorDimension. As it is subclassed, the attributes and methods listed are additional ones compared to Dimension. When initialized, this Dimension should be fed the dimension it was based off of, in addition to a Locator, information about whether the dimension is to be created or collapsed, and the axis of the dimension. The main unique method to a Locator Dimension is a way to transform the LocatorDimension back into its parent dimension.

**E.4.3.6: LocatorDimSet**

A LocatorDimSet (short for “Locator Dimension Set”) is a container of Locator Dimensions. This class is not subclassed from DimSet. One LocatorDimSet is used for each Locator, so only those Dimensions associated with the Locator are stored within a given LocatorDimSet. The LocatorDimSet contains a list of dimensions to be created or destroyed, as well as its corresponding locator and a link back to the DimSet it is a part of. The methods include ways to slice, a way to create templates for an output dataset, and most importantly, a way to perform the locator associated with the LocatorDimSet on an input DimSet, creating the slice object and dimension needed for DataSelector task extraction. LocatorDimSets, stored within DimSets, are typically iterated over by DataSelector tasks.

**E.4.4: Data Selector Classes**

As DataView is a visual program handling generic containers of labelled data, the indexing routines used in NumPy are not sufficient enough to handle data manipulation. For this, we use
the DataSelector and its corresponding tasks. The tasks in the documentation are those at the time of the writing, but it is a modular system much like the more complex Method and Viewer systems.

**E.4.4.1: DataSelector**

A DataSelector is an object composed of a series of tasks which slice and dice a DataSet in many different ways to view a subset of data. A DataSelector is as its heart a DataSet, with the important caveat that it is built on top of another DataSet (its “input”) through a series of Data Selector Tasks (dsTasks). It stores these tasks in a doubly linked list\(^{159}\), connecting a head task, from which each task is in turn connected to the next task. It is possible to iterate both down and up the list of tasks. A DataSelector must be processed, which initiates the processing of the input DataSet from the beginning of the task chain. It is the backbone of the connection between the data and Viewers there are an assortment of helper methods for common tasks useful for viewers, such as extracting a list of locators, retrieving a value of a processed DataSet, and obtaining the minimum and maximum values of a proportion of the DataSet (useful when setting plot scales). The DataSelector contains a PyQt signal called “processUpdated” which is called whenever a Locator connected to one of the tasks is triggered, to trigger the processing of the input DataSet. It also contains a History, which contains a separate signal when the underlying DataSet is modified.

**E.4.4.2: Data Selector Task Classes**

A DataSelector is composed of a linked list of tasks. These tasks are registered in a modular system. The Data Selector Task is an object which defines how to pick a particular subset of a data object. Unlike the specific DataSelector to a DataSet object, it is not tied to any specific data object. A Task will work on any object with the same number of required Dimensions. The `DSTaskBase` is the base class for all DSTasks. DsTasks keep track of where their dataset should be coming from, and where it is going to—other tasks, the input, or the output. They have some parameters which determine how they act. They also have a “process” method which is called by their predecessor, which they then pass off the results of to their follower. They also have class constants, overwritten in each derived task class, to determine the editable and reshapable properties of the tasks. The editable property determines whether an editable dataset will remain editable after processed by a task; routines such as statistics do not keep editability, but those routines that slice such as selection do. This tells us if the underlying array is stored elsewhere in memory as a “view.” The reshapable property determines whether the task inherently subsets the data, returning only a portion of the dataset. If it does, then processing routines which change the data shape, such as cropping, will not work because they will only
change the shape of a portion of the original dataset. Locator routines do not inherently subset because although they do, they can be accessed by iterating over the locator index.

**E.4.4.3: Select Task (DSTaskSelect)**

The Select Task is the simplest task, collapsing a set of Dimensions by choosing a single element of them. It is reshappable and editable. Processing the dimset and dataset are trivial, as they call the underlying slice routines. This is a good simple example of how a Data Selector task can be constructed.

**E.4.4.4: Statistics Task (DSTaskStatistics)**

The Statistics Task collapses one or more Dimensions using NumPy statistical methods. All methods handled do not change the units of the data. Statistics methods handled by this task include mean, median, standard deviation, minimum, maximum, sum, and range (maximum - minimum). Statistics maintains neither reshapability nor editability.

**E.4.4.5: Transpose Task (DSTaskTranspose)**

The Transpose Task transposes the dataset. It is a simple method much like the select task, being a wrapper on the underlying transpose methods for DimSets and DataSets. Transpose maintains reshapability and editability, so it creates a view in the underlying NumPy array.

**E.4.4.6: Meld Task (DSTaskMeld)**

The Meld Task reshapes the dataset by joining together two or more Dimensions into a single dimension. Necessary parameters include an ordered list of dimension names to meld, and the name of the new dimension to create, by default being unitless. Meld maintains reshapability and editability, so it creates a view in the underlying NumPy array.
E.4.4.7: Locator Task (DSTaskLocator)

The Locator Task simply connects Dimensions in an incoming DimSet with Dimensions in a locator. It turns these Dimensions into LocatorDimensions. It also creates any new LocatorDimensions. The shape of the DataSet remains the same. As the creation of a locator does nothing to change the data, it maintains editability and reshapability. It is the handling of locators that changes these properties.

E.4.4.8: Locator Handler Task (DSTaskLocatorHandler)

The Locator Handler Task handles one or more locators in a DataSet. The locators will have previously been defined in a Locator Task. This task will store the unhandled dataset and will subscribe to it, so that when locators change it will immediately initiate the process chain to push through the new dataset values. Typically this will be called just once at the end of the Data Selector task chain in order to handle Pickers, PickerArrays and PickerClusters. However, if a task wants to act on a locator generated dimension then this task must come before that - ideally just before, as the longer you wait to actualize locators the better. There is no need to handle Picker locators in the middle of the chain, as they as they do not create new Dimensions that might be acted on. However, the very last step in the chain will be a LocatorHandler to take care of pickers. LocatorHandlers simply slice the data, so it creates a view and maintains editability. However, by knocking certain dimensions out of the original dataset, this subset cannot be reshaped.

E.4.4.9: Unlocator Task (DSTaskUnlocator)

The Unlocator Task disconnects Dimensions in an incoming DimSet from a Locator with which they have previously been connected. It turns LocatorDimensions back into regular Dimensions. This is primarily used by Datalterators, which may need to iterate over a picked Dimension. As this changes an underlying class and doesn’t change the data itself, it maintains editability and reshapability.

E.4.5: Data Object Chooser

A Data Object Chooser is an object which takes initial information about input datasets from some front-end object, as well as the method the chooser will work with, to create input for a Datalterator that is consistent with the way the method works. It is composed of a small
wrapper class as well as some GUI widget classes for the front-end. The GUI widget is only called if the method requires more than one dataset, or more than one dataset selected as input; the dataset is passed through otherwise. Data Object Choosers select the DataSets and sets the contexts of the input datasets for the method.

A context is a list of what to do for each of the dataset’s axis Dimensions. A Dimension can be iterated over, fixed to a specific index, picked to the current index if the axis has a picker associated with it, or simply passed through. When iterated over or fixed, the context changes the dimensionality, destroying the dimension in the DataIterator.

The GUI allows the user to select any currently open DataSelector or DataSet in the program as an input. DataSelectors selected are converted into DataSets for the input. When setting an input, the dataset and its context must be set in such a way that the input is valid for the method. For example, a method might require the first input have two dimensions, and the second input as having the same dimensionality of the first input. These requirements are set in a method’s parameters.

**E.4.6: DataIterator**

The DataIterator class is the construct used by methods to generically iterate over DataSets. A DataIterator informs methods how to think about the data it is manipulating by retooling a DataSet so that the method can operate on subsets of the data. They generically act on DataSets, but are typically applied on DataSelectors by methods. A DataIterator can be considered a wrapper over a flexible iterator acting on NumPy arrays. It can also be thought as a DataSelector itself, because it modifies the way that objects look at data, just like a DataSelector does for viewers.

A DataIterator requires two lists of dictionaries as input: Dataset Parameters and Method Parameters. This is necessary as methods may act on one or more DataSets and information is needed to describe the input dataset. Each element of the list corresponds to parameters referring to one input dataset. The Dataset Parameters are parameters describing the actual DataSet itself, and importantly contains a context, to be described later. The DataSet Parameters are created by the Data Object Chooser, explained in section 3.4.6. The Method Parameters are parameters taken from Methods, describing properties of the input dataset such as necessary requirements for validation and whether the object is to be chunked over while iterating.

Knowing when to iterate over a dimension or passing through a dimension is important when using methods, because the underlying functions typically applied with NumPy or SciPy may apply on a subset of the axes or all of the axes of an underlying array. For example, smoothing over all axes of a three dimensional DataSet has a different effect than smoothing only in one dimension. In addition to the result, the context is important in the performance of a method’s
algorithm – passing through axes is typically faster than iterating over them. This is because in NumPy, the former acts on a large array and the iteration is done at C speeds, while the latter acts on smaller arrays, the iteration being done at python speeds – python is a slower language than C. The DataIterator object is planned to be rewritten in Cython, an optimizing static compiler in an extended version of Python that allows for the calling of C functions and C types\textsuperscript{161} to optimize the speed of the code.

Another important attribute for a DataIterator is its ability to use chunks, which determines if a DataIterator will iterate over multiple layers at a time (a “chunk”) or a single one. At the low level, a DataIterator creates another dimension to iterate over and reshapes all iterated dimensions into this axis. If a DataIterator uses chunks, it takes slices of this iterator dimension at a time depending, the size of which depends on memory thresholds – if not, it iterates over one index of the iterator dimension at a time.

A DataIterator can have multiple inputs and output DataSets, and can modify already existing DataSets. New datasets are created using a DataIterator’s create method, which creates a DataSet whose array is originally blanked. The create method requires a DimSet of all non-iterated dimensions. It also requires the name of the key of the dataset which shares the same iterated dimensions, which also determines when the new dataset is iterated over. All input and output datasets must have their own unique keys, which can be accessed by the create and update methods.

To apply a DataIterator on DataSets, the DataIterator must be first created; typically this is a result of passing through inputs through the Data Object Chooser mentioned in the previous section. Creating new DataSets requires the use of the create method with unique names in the Method. When iterating over a DataIterator, the DataIterator yields the appropriate smaller subarray that the method calls for. To place the data into the output array, whether that is by modifying the original array or the new arrays, you need to use the update method which makes changes to the appropriate DataSet, targeting a specific stored DataIterator. When updating the DataIterator, the injected array needs to have the correct dimensionality of the DataSet it is being placed into.

A DataIterator can keep track of multiple iterations happening within the object, to allow for iterating over more than one DataSet simultaneously – for example, when one needs to modify two different DataSets at once. Control of the iteration in a DataIterator is done by key access to determine which dataset to iterate over. DataSets can share the same iteration by the method noting that they are linked in the method parameters – if this is the case, then the two DataSets must share the same iterated dimensions.

The following code gives examples for multiple use cases of a DataIterator within a Method.

DataIterator which updates a single input (‘in0’) DataSet, e.g. in a process method:

```python
>>> # no creation necessary
>>> for array in DI: # Iterate over DI, array is a numpy view of part of the DataSet
```
DataIterator with one input ('in0') and one created output ('out0') DataSet, e.g. in an analyze method:

```python
>>> DI.create('out0', dimset, 'in0')  # Creation of an output DataSet out0 sharing iteration with in0
>>> for array in DI['in0']:
...   DI.update(func(array), name='out0')  # Apply update over output DataSet
... dataset = DI.output('out0')  # Spit out output
```

DataIterator with one input ('in0') and two created output ('out0', 'out1') DataSets, e.g. in an analyze method:

```python
>>> DI.create('out0', dimset0, 'in0')  # Create two stored DIs and output DataSets
>>> DI.create('out1', dimset1, 'in0')  # shares same iterated dimensions as input
>>> for array in DI['in0']:
...   DI.update(func1(array), name='out0')  # Apply update over first output DataSet using one function over an array
...   DI.update(func2(array), name='out1')  # Apply update over second output DataSet using a second function over an array
... dataset0 = DI.output['out0']  # First output DataSet, accessed through its name
... dataset1 = DI.output['out1']  # Second output DataSet, accessed through its name
```

DataIterator with three inputs ('in0' and 'in1' linked, 'in2' separate) and two created output ('out0', 'out1') DataSets:

```python
>>> DI.create('out0', dimset0, 'in0')  # Create two stored DIs and output DataSets
>>> DI.create('out1', dimset1, 'in1')  # shares same iterated dimensions as input
>>> for array1, array2 in DI['in0', 'in1']  # These two DataSets are linked
...   DI.update(func1(array1, array2), name='out0')  # can work with 'in0' and 'in1' data
>>> for array3 in DI['in2']:
...   DI.update(func2(array1, array2, array3), name='out1')  # can access all three datasets
```
E.5: Main Classes

The main package contains a number of core classes critical for the operation of the rest of the program. These classes typically glue different components of DataView together, connecting aspects of the Data together with the modular systems which the software is composed of.

E.5.1: Registration System

DataView is built upon its registration system for its modular components. The registration system is built on a number of components, including a metaclass system tied within a base class, and a series of register module functions for each modular subpackage. The registration system is used for Data Selector Tasks, File Handlers, Locator Widgets, Methods, Simulators, and Viewers.

The main entry point for DataView, \texttt{dataview.main.dv}, contains all the registration functions for the modules. Each registration function imports all submodules of a corresponding package recursively, including subpackages. Importing the submodules works with the metaclass system to register all of the inherited classes of a module’s base class; this is stored inside the base class.

In python, a metaclass is a “class” of a “class” – like a class defines how an instance of a class behaves, a metaclass defines how a class behaves. A class is an instance of a metaclass. Each modular system has a specific metaclass tied to it, with similar methods. Each metaclass subclasses Python’s ABCMeta class, part of python’s \texttt{abc} module\textsuperscript{161}. The metaclass system used by the modules allow abstract base class decoration and implements record keeping in classes created by it, enabling auto-registration of all classes in a subpackage.

The base class of a module is inherited by all submodules, such as specific viewers, methods or file handlers. A base class serves as an abstract base class for all objects of its type. An abstract base class is a form of interface which ensures that derived classes implement particular methods from a class. For example, a File Handler needs to implement methods like load or save, because all File Handlers need to be able to handle these methods when operated by the program. Base classes may have functions which call classes which inherit from it, so instead of importing a specific module in a submodule when grabbing a class or creating an instance of that class, you just need to import the base class and call the method which grabs the class or creates an instance. It is also possible to create base classes which derive from previous base classes for further templating – for example, \texttt{MPLViewerBase} is a base class for Viewers which
are objects which display data on a MatPlotLib class, which in turn inherits from ViewerBase, the base class of all Viewers.

Registering classes in this system is useful for the software to dynamically handle systems such as preferences, menus, and actions, as extra information needed for these systems can be stored as class attributes that are easily modifiable by module programmers of DataView. This dynamic system allows programmers to create new classes without having to modify all of the necessary modules that need to be attached to them.

E.5.2: Action System

The Action System, which consists of the DVAction and DVProcessAction classes (distinguished from PyQt’s QAction system), is an essential aspect of DataView’s Method system. A DataView Method class is distinguished from a Python class method the in that the former is a class subclassed from the MethodBase abstract base class, and modifies or creates new data, while the latter is simply a function of a class. This documentation will distinguish the two by calling a class method a class procedure instead of the usual python terminology.

A DVAction contains all the necessary information to call a given method. Every method class must have an “execute” procedure which takes as input a recipient, a dictionary of “action information”, and a DataIterator. A recipient contains information about which data to act on, typically either a DataSelector if operating on data, or a Viewer for displaying methods. The dictionary of action information contains any information specific to the action such as the bounds of a crop, the parameters of a plane to subtract, etc. The DataIterator is the object used to iterate over the recipient, and also contains information about how the action was called, such as what kind of display it was called from, and how to iterate over it via the context. Both of the latter pieces of information, the action info and the DataIterator, are contained within the DVAction class. This allows automation of method calls, such as the construction of macros, as well as automation of histories, and the undo/redo system.

A DVAction contains attributes such as the method class corresponding to it, a short and long human readable description of the action, information particular to the method such as crop coordinates, the DataIterator attached to the action, the name of the user, and the start and stop times for the action execution used for the history. It also contains an “execute” procedure, which calls a Method’s execute procedure and adds the action to the history system.

A DVProcessAction is a modified DVAction (inheriting from the respective class) designed to handle actions which process the recipient, modifying it. Unlike DVActions, DVProcessActions need to handle undos and redos, adding an additional attribute to the class for the undo system which in turn is a DVProcessAction, specifying the action needed to undo the action. For example, if one method is ‘add’, its undo method would be ‘subtract’. A common undo method is UndoFromFile, which will be used for all processes that are not invertible.
E.5.3: Menu System

Like other aspects of the graphical user interface (GUI) in DataView, the menu system is built up using the Qt Framework using PyQt. A naïve system would generate a menu system statically—the elements of the menu generated in the same submodule as the construction of the menu. The problem with this is the amount of repetition needed when constructing new parts of the framework. For example, when adding a Method module, you would have to add all references of the method to different parts of the menu to allow users to use the Method. This is inefficient, and we would prefer to have a system that is more “plug and play”. So in addition to the Qt interface for the menu system, composed of the QAction and QMenu classes, DataView has added classes for an interface for a list-based system to generate the equivalent Qt classes from attributes of the classes which need to be added to menus, the DVMenuItem and DVMenu.

Qt's QAction class provides an abstract user interface action that can be added to widgets. Many common commands can be involved via menus, toolbar buttons, or keyboard shortcuts. Each command should be represented in the same way—thus, Qt provides the QAction class to represent the command as an “action”. This is not to be confused with DataView's DVAAction classes, which are not similar in form or function to Qt's QAction class. A QAction is added to menus and toolbars and are automatically kept in sync. A QAction is, for example, an individual menu item in a QMenu, such as a “save” item in the menu. A QAction may contain an icon, menu text, shortcut, status text, tooltip, and “What’s this” text, all corresponding to different ways to view and act on menu items or toolbar buttons.

Qt's QMenu class provides a menu widget for use in menu bars, context menus, and other popup menus. A context menu is a menu that pops up when a user right clicks on an object. In DataView, context menus are often used in Viewers to show the different ways one can act on the data. A QMenu contains one or more QAction objects, or cascaded QMenu objects—so a menu can contain menu items or other menu which in turn has its own menu items.

DataView has its own classes that are equivalent to Qt’s QAction and QMenu classes. One of these classes is the DVMenuItem. This is equivalent to the QAction class, and is used to create template menu actions that are then converted into QActions where the QMenu is generated, such as when a new display is created. The template consists of a dictionary with key parameters that are attributes of a QAction and this is used in modules (such as methods) to generate a part of the menu. A DVMenuItem has a procedure to set parameters and a procedure to convert the template to a QAction. The following is an example of a DVMenuItem template:

```python
>>> {'text': 'Hello World', 'tip': 'Demo action item'}
```

Here, “text” defines the text displayed by the menu item, and “tip” displays the tooltip displayed when the menu item has a cursor placed over it.
The DVMenu is a python container (dictionary, list, and text) based version of Qt’s QMenu class. It allows easy storage of the menu in a configuration file, as well as easy hand-coding for menus by users. The DVMenu is initialized with a nested list of menu items. Every list has the format [MenuName, item, item, ...]. Items can either be other menus, which are lists of the same format, or they can be text, or a dictionary with more information about the action, such as shortcuts. This dictionary is the template inside a DVMenuItem. The DVMenu is a subclass of a Python list. It has a procedure to add a list of items to the menu, and a procedure to append a new menu to a QMenu. Figure 5-5 shows an example of how a DVMenu’s nested list of menu items correspond to the GUI’s QMenu.

![DVMenu Example](image)

Figure E-3: DVMenu Example. Example of how a text-based nested list of items, stored in a DVMenu, corresponds to a GUI QMenu.

### E.5.4: Unit Registry

Pint is the package used by DataView to define units. Pint has a concept of a Unit Registry, which is an object with within units are defined and handled. When using units, the Unit Registry object needs to be instanced. This populates the registry with the default units stored within Pint, but we might need to define our own units; these can be defined later after creating the registry. DataView holds a module which defines the unit registry for the rest of the software. All a programmer needs to do is import the unit registry from this script, and the units can be used elsewhere in other modules.

This module also defines Quantities. A Quantity is a product of a numerical value and a unit of measurement, and in Pint, is defined by its magnitude, units, and dimensionality. It can handle mathematical operations between other Quantities – for example, you can define speed by dividing a distance quantity by a time quantity. The unit registry knows about the relationship between different units, and we can convert quantities to the unit of choice. Units in DataView, such as those within Dimensions, are stored in terms of Quantities for easy mathematical operation.
The module also defines the “pixel” unit as a new fundamental unit in the Unit Registry. Its relationship with other units is defined via unit contexts (different from contexts in DataIterators or context menus in the GUI). There are two different types of pixels: “delta_pixels”, which only make sense when the distance from one pixel to another is the same, and “absolute_pixels”, which can just use the Converter routines.

E.5.5: History System

DataView’s history system is composed of its History and Undo classes. These are objects which operate on DVActions. They keep a record of what is done to the data, and enables undoing and redoing of methods on those actions.

A History contains a list of actions which can contain multiple branches of actions. If the list is branched due to the use of undo, then the history splits into two lists, where the first list contains the new actions, and the second list contains the old actions before the undo. A History is a Qt QObject, so we can use QT’s signal and slot system to trigger procedures on an object when a History is changed. A History contains an attribute that contains the actual full branched list of actions, as well as the current sublist being acted on. We can also dump out the History in different formats, such as a list of DVActions that are currently undoable.

The history system is also composed of the Undo method, which handles undo and redo processes. Like all method classes, Undo has only class methods, and never needs to be instanced. Individual “undos” are contained by DVProcessActions, which are listed in a History. Undo also stores procedures which backs up the data before it is modified as a temporary file stored in DataView’s HDF5 format.

E.5.6: Object Reference and Naming System

A number of DataView objects hold lists of the instances of the classes which exist. To handle this, the DVWeakMemory class was created, which is a wrapper on Python’s Weak Value Dictionary. Weak Value Dictionaries are used throughout DataView to keep collections of objects without preventing those objects from being garbage collected if they are deleted elsewhere. This removes the requirement that the collection be notified when we want to destroy the object. For example, many classes have the class attribute _objectList, which contains all (non-temporary) objects of that class, allowing us to easily iterate over them, for example. The wrapper was created to allow the software to handle serialization into the native HDF5 format.

A number of DataView objects, such as most data objects, collections, and viewers have names. In the context of the program, they need to have unique names when they are stored as
references in other objects. For example, a DataSet holds a container which reference all the DataSelectors that use it as an input, and a container for all DVCollections that hold it.

The DVName class handles the assignment of unique names to dataview objects. With DVCollection and DVWeakMemory, in order to effectively use these classes, we must have unique names. A DVName assures that the name assigned to an object is unique. It enforces a naming standard of /owner.name/object.name where the “owner” of an object, when one exists, is the single object to which this object belongs. For example, DataSelectors are owned by DataSets. Viewers are owned by their ViewObjects. DVCollections don’t have owners and neither do DataSets. However, both can be in collections, but that isn’t considered ownership. As a rule, an object’s owner can never change. This is important, because if it did, then the object’s full name would need to change.

E.5.7: Logging and Macro System

DataView’s logging system is composed of a module which contains routines for log creation, using Python’s “logger” package. There are two main uses for logs. The first use of logging is debugging. Information about the code is dumped to a log file and/or the console, depending on the debug level. The level of the logger is set at the top level of the program. Multiple debug loggers can be used, in case you want to debug a section of the code separately. They are identified by a globally accessed name, such as “root” for the root logger.

Logging is also used for logging actions. This is standardized so that log files can be easily and effectively used as macros. The macro system is not currently implemented, although there is code for a parser of macro code developed.

A logger is used by modules by first importing the logger and setting the logger to a variable. After this, all you need to do is the following:

```python
>>> rootlog.level("message")
```

Here, “rootlog” is the name of the log variable, “level” is the debug level of the message, and the string inside the function is the text of the message. This is similar in form to printing the message of the console, except the debug level needs to be set when assigning the message. The debug levels available are “debug”, “info”, “warning”, “error”, and “critical” in order of increasingly important levels.

E.5.8: Login and Preferences System

DataView has a preference system composed of a DVPreference class, which handles interaction with initialization (or preference) files, which are held in a separate folder. The DVPreference class reads initialization files by default when the program starts. Initialization files are written
to be used with the “ConfigObj 4” python package. This is a simple but powerful config file reader and writer with many features.

There are two typical preference files in DataView. The main preference file is named “DVPref.ini”. This is stored in the preference directory, located in the dataview source directory. In addition, each user may have a number of different preference initialization files. This allows customization based on the type of data analysis being done – you might want to only use a subset of methods in the program at any given time. These initialization files are named according to the username and “profile” as “username_profile.ini”. The “username” in the file name is stripped to alphanumeric characters only, and lowercase. These user preference files may be located anywhere, including online. Where these files are located are stored within DVPref.ini. By default, the user preference files are also stored within the preferences directory. On the GitHub, these initialization files are not stored – DVPref.ini is automatically generated when the program first starts and new user profiles are created by copying other profiles.

The front end of the preferences system is the DVLogin class, which creates a login box for the user to select their name and profile. Figure 5-6 shows a visualization of this system.

Figure E-4: Log in Screen of DataView, with the selections taken from the preference system.

E.6: File Handlers

To handle the many different file formats that DataView will be able to handle, DataView contains a modular system to load and write from and to data files. The filehandlers package contains all of these File Handler modules.

E.6.1: Structure of a File Handler

File handlers are subclassed from the FileHandlerBase class which defines the structure of all File Handlers. File handlers are never instantiated, and only contain class methods. Class methods include loading from a file, writing to a file, configurations for the file, and how to display the
data by default. A file handler may also contain additional helper methods which aid in the file handling process.

A File handler contains two class attributes. The first class attribute is a dictionary storing information about the file types called FILETYPES. This is the list of allowed filetypes, which describe the nature of the data stored in the file. For example, the file could hold one or two dimensional DataSets, or collections of DataSets, or text information and macros.

The second class attribute is an information dictionary, which is the only attribute which must be edited in each class. It contains information about the version of the file handler, a user readable set of information about the handler, and a list of handled extensions (e.g. ‘jpg’ or ‘hdl’). It contains filters, which are a set of user defined categories into which we might sort the type of file handled by the handler. For example, JPG and TIF might be ‘Images’, while SXM files might be ‘Topographies’ and ‘STM Files’. It also contains a boolean to determine if we need extra configuration information to save the file – for example, a JPG might need to set its quality before saving. It also contains a set of types of data that can be written to, that is a subset of the FILETYPES. The last part of this dictionary is a boolean to determine if the program is enabled to read these files.

The load class method is a generic method which loads a single file referred to by its filename. The functionality of the load class method can be very different depending on the nature of the data being converted. If the file handler creates DataSets, then typically a header is read from the file to determine the nature of the data stored, and the data is obtained as one or more raw NumPy arrays. A DataSet is built from the ground up; the arrays are stored in DataBlocks, and DimSets are created from Dimensions determined by the nature of the data. The method would also create DataSelectors to view the data in different ways that would be useful for viewers later on. Regardless of the nature of the processed data, all of the data objects are stored in a DVCollection.

The display class method is used to create a number of default viewers to view the data processed as data objects. It takes the DVCollection returned from the load method, and typically creates a new ViewGroup to store the viewers, and a Tree Viewer to view the hierarchy of the data extracted from the file. Typically they would take the DataSelectors created in the load class method, store them in viewers, and build the LocatorWidgets from locators stored in the DataSelectors for these viewers. There is also a load_display class method, which typically doesn’t change depending on the file handler, which simply loads the file and returns the file’s collection to the file handler’s display method.

The save class method is writes data to the file format. The class method takes the dataview object, filetype (such as ‘data’ or ‘viewer’) and a dictionary of configuration information and writes the file using a class-specific way of writing the data. Not all file formats can be saved; some are read only, such as Nanonis files.

The get_configuration class method generates a configuration dictionary with additional information about how to save data – such as setting the Quality in a JPEG file. The main file
handler will maintain a list of all file handlers and the data and viewer information they have saved, indexing the last save configuration. This will be used if the same information is saved again, unless the user specifically requests to change the configuration. In that case, or in the case where information has not yet been saved in this format, this method will be called to generate the configuration, typically via a GUI.

E.6.2: Structure of the native HDF format

DataView’s native file format is the HDF5 binary file format, a versatile data model that can represent very complex data objects and a wide variety of metadata. While the HDF5 file format can store arbitrary hierarchies of arrays, DataView uses the format in a specific fashion. A typical file handler, such as the Nanonis file format, opens the data in an organized fashion, knowing how many data objects are going to be created and the typical viewers and locaters to be displayed upon opening the file format. The native format makes no assumptions about the structure of the data stored. If there are viewers stored in the file, the program opens them, and if there isn’t, it doesn’t.

Any data object (e.g. DataBlock, DataSet, or DataSelector), dimension object, viewer, or collection can be saved either individually or including subobjects. For example, DataBlocks can be saved by themselves or as part of DataSets. HDF5 is a hierarchical data format, which uses groups to organize ‘datasets’ which can be labelled with ‘attributes’. In the data format, every class is stored as a group. The type of class is stored in a group attribute "classtype." Parameters which are classes become subgroups, while ‘simple’ parameters become either datasets or attributes.

The group/dataset/attribute names are the parameter names.

Every dataview object that can be stored in an HDF file has its own class methods to read and write from and to an HDF file. As attributes are very constrained in HDF5 files, if an attribute is too complex to be stored, they are serialized as a byte array using python’s pickle object serialization process.

E.7: Viewer and Widget Classes

The viewers package stores all classes related to the modular GUI portions of the software; the most important parts of which are Viewers, ViewGroups, and LocatorWidgets. Figure 5-5 illustrates what these different kinds of objects look like on the front-end of the software. Development of all of these objects requires knowledge of programming Qt in python using the PyQt package.
E.7.1: Viewers

Viewers are objects which display the data in the program. Most typically, they view a slice of a data through the use of a DataSelector, but other viewers, such as TreeViewers, work on other objects, such as DVCollections. Viewers are a modular system, having a metaclass for registration like other modular systems in the program, with a base class called ViewerBase which all Viewers are derived from.

E.7.1.1: ViewerBase

The ViewerBase is subclassed from PyQt’s QMdiSubWindow class, which provides a subwindow for an area in which MDI (multiple document interface) windows are displayed. It serves as an abstract base class for all Viewers. It handles a lot of the complex initialization code required to integrate attached objects (called View Objects in the code) and Locator Widgets into the window. It handles the creation of the context menu that one can use by right clicking a widget, as well as the code required to read and write the Viewer to HDF format. It also handles the passing methods selected from the context method, passing the viewer’s View Object as input into a method, and the push of the Data and method parameters to a Data Object Chooser and ultimately a DataIterator. As a result of all the complexity done in this class, Viewers created by the programmer are far easier to create. The ViewerBase also holds a list of all instantiated Viewers that can be iterated over. There are a number of abstract methods that this class has, which will be described in the next section.
E.7.1.2: Structure of Viewers

When a programmer creates a new Viewer by subclassing ViewerBase, it has to follow a specific structure. This includes class attributes that need to be detailed and an abstract method that must be declared.

There are two class attributes of a Viewer. The first class attribute is the information dictionary. This contains information about the version of the viewer, which is important as it is used to determine whether menus need to be regenerated and whether action lists can be run in the same fashion, such as within automated Methods.

The information dictionary also contains the viewclass, which is a class selected by the programmer. Each viewer type can display only one class of object. For example, Image Viewers and Plots will view DataSelectors, while a property viewer might view a viewer. In the case that...
the viewer should be able to view more than one kind of object then superclass the two objects and view that.

The information dictionary finally contains \textit{constraints}, a dictionary. To see if a specific viewclass object is suitable for viewing in this viewer, each of the method keys in this dictionary will be called and checked against the given values. The viewer will only work if the values match. For example, a viewer might only work on 2D data, so it would be

\{'numDimensions': 2\}.

The second class attribute is the \textit{displaymenu}, a dictionary containing the top half of the “display” menu in the context menu system. These are Methods specific to this viewer. Menus can be defined by simple text, by a dictionary with any subset of the following keys, which can be abbreviated: ['text', 'icon', 'shortcut', 'tip', 'checked', 'whatsThis', 'name']. They can also have lists of multiple items, or nested lists to make submenus. Some examples:

'Show error bars...' # a simple string menu item
\{ 'te': 'Show error bars...', 'sh': 'Ctrl+B' \} # dictionary defined

Viewers are based on a core widget which is to be assigned on the viewer, as well as a viewObject, which is the data object to be viewed using the widget. The nature of taking the data and viewing it is highly viewer dependent. Some viewers will be based off of Matplotlib; these viewers have an intermediate base class called MPLViewerBase which adds code necessary for Matplotlib functionality. Other viewers, such as History Viewers, Property Viewers, or Tree Viewers, use Qt’s widgets to view aspects of data.

The abstract method that needs to be set is the \textit{setup} method. This method sets up the UI of the Viewer. After the specific details of the customized Viewer’s setup, the method needs to call \texttt{super().setup()} to set up the menus from the super class’s initialization. In the setup method, if a viewObject has a history and the data can be modified by a Method, the viewObject needs to connect the history’s signals to slots created in the Viewer. For example, here is a setup function for a Matplotlib Viewer, whose viewObject is a DataSelector:

\begin{verbatim}
def setup(self):
    MPLViewerBase.setup(self)
    self.plot() # function to create a plot
    # Replot when the dataset is changed by a method
    self.viewObject.history.connect(self.replot)
    # Refresh when dataselector pickers change
    self.viewObject.connect_process(self.refresh)
\end{verbatim}
Here, replot and refresh are slot functions defined in the Viewer’s methods. Separate functions are needed because drawing an image for the first time is more computationally intensive than updating an image. It calls the super class of the viewer, MPLViewerBase, which in turn defines setup information important for all Matplotlib Viewers.

### E.7.2: ViewGroups

A **ViewGroup** is a window which serves as a GUI container to hold different viewers. Typically when a file is opened, viewers relevant to viewing the data stored in the file are all held in one ViewGroup. The class is a wrapper around Qt’s QMdiSubWindow, whose widget is a QMdiArea, which provides an area for multiple document interface windows to be displayed – in our case, viewers.

The most important functionality of a ViewGroup is that when the user wants to add a Viewer to a ViewGroup, they can use the *add* method to create the class by string name. There is no need to import the individual class name of the Viewer due to the viewer registration system. ViewGroups can also be written to and from HDF files like many other DataView Objects.

### E.7.3: LocatorWidgets

A **LocatorWidget** is a wrapper class on Widgets which are connected to Locators. This is designed as a wrapper class rather than a mixin class on QWidgets due to an issue with PyQt 5 with QObjects having mixins with metaclasses. A LocatorWidget uses the modular class system, having a base class LocatorWidgetBase with a registration system created by a metaclass. It has a template structure much like other modular classes.

There are three class attributes. The first is an information dictionary which stores information about the version of the LocatorWidget, and the class of the Locator that is to be stored in the LocatorWidget. The second attribute is *dimensions*, an integer number determining the number of dimensions stored in the Locator for this widget. The third attribute is the *widgetType*, which is the QWidget class that the LocatorWidget is wrapped around. For example, the LWComboBox uses a QComboBox with a Picker locator, acting only on one dimension.

The first abstract method that needs to be defined is *setup*, which at the least sets the widget attribute as an instantiated widgetType, and connects the slot of the Locator Widget to the widget. The second abstract method is *postprocess*, which is essentially a second setup method, to be acted on after the Viewer has been displayed. This is useful if something needs to be drawn after the Viewer has been displayed, such as a cursor.
The third abstract method is the *slot*. This is a function which updates the index of the LocatorWidget’s locator. This is connected to the third abstract method, *connect*, which connects the slot to the signal of the widget.

Some LocatorWidgets are not related to QWidgets, but are hand constructed. For example, MPLCursors are cursors built on top of Matplotlib. All the drawing and extra signals and slots needed to create the functionality of a cursor on a Matplotlib object (as seen in Matplotlib) are created as extra methods here.

### E.7.4: Example of setting up ViewGroups, Viewers, and LocatorWidgets

The following shows example code for the interface to create a ViewGroup, a Viewer from a DataSelector, and creating Cursor and ComboBox LocatorWidgets for this Viewer from the pickers stored in the DataSelector.

```python
>>> # Primary DataSelector to work with
>>> image = dataset_experiment.my_dataselectors['Image']
>>> # DataSelector that contains the cursor picker
>>> curve = dataset_experiment.my_dataselectors['Curve']
>>> exp_xy = curve.get_locator('Cr-xy')  # cursor picker
>>> # Create ViewGroup
>>> viewgroup = ViewGroup(title=filename)
>>> # Create Viewer
>>> experimentV = viewgroup.add_viewer('ImgViewer', image)
>>> # Create Combo Boxes - these are names of the pickers for combo boxes
>>> for name in ['Cb-dt', 'Cb-r', 'Cb-fb', 'Cb-E']:
>>>     experimentV.addLWidget('ComboBox', image.get_locator(name), key='comboboxes')
>>> # Create Cursor
>>> experimentV.addLWidget('MPLCursor', exp_xy, name='XY-EXP', key='cursor')
>>> # Display
>>> experimentV.display()
```

### E.7.5: Widgets

Widgets stored in the *viewers/widgets* directory include all other accessory widgets used as parts of Viewers which are not LocatorWidgets. These widgets lack a modular structure like
LocatorWidgets, and there is no specific template structure to them. An example is a
*ComboBox*, which stores all of the ComboBox LocatorWidgets in a general fashion for
viewers which use ComboBoxes. Another example is a *FloatingToolbar*, which generates a
template for the toolbars used by the program. For example, there can be a floating toolbar to
display different ways of selecting regions of interest.

**E.8: Methods**

Methods are routines which analyze, process, or display data. They may modify existing data or
create new data. The method system uses the modular class system extensively. All methods
are subclassed from the *MethodBase* class. Methods are never instantiated – all of their
routines are called upon as class functions.

**E.8.1: Structure of all Methods**

All Methods have a number of class attributes that need to be set and abstract routines that
need to be detailed.

*Info* is a class attribute; a dictionary which contains information about the Method. Like
elsewhere it contains the *version* of the class. Specific to methods, it contains *submenu*, which is
used to group together methods in a single submenu in the context menu. It can be a blank
string (''), in which case the menu item(s) will be in the main part of the relevant menu, such as
under the Process or Analyze menus depending on the method. It can also be a menu name (e.g.
‘Special’) which will create the submenu and put this and any other methods which list the same
submenu in it, or a submenu structure (e.g. ‘Special.2015’) in which case a nested submenu
structure will be created.

*Menus* is a dictionary class attribute which is a collection of menus keyed by the `vista` (see
`VISTAS`) in which that menu is to be used. The menu format is flexible and is the DVMenu
format. Menus (for DVMenu) can be defined by simple text, by a dictionary with any subset of
the following keys (you can abbreviate them):

```
['text','icon','shortcut','tip','checked','whatsThis','name']
```

or with lists of multiple items or nested lists to make submenus. Note that with lists that the first
item is the menu name and the other items show up as a sublist. The following are some
examples of submenus:

```
{ '1D' : 'Line subtraction...' } # a simple string menu item
{ '1D' : { 'te':'Line subtraction...', 'sh':'Ctrl+L' } } # dict defined
{ '2D' : [ 'Background Subtraction', 'Plane', '2nd order' ] }
```
{ '1D' : 'Line sub', '2D' : 'Plane sub'}

**UserExplanation** is a class attribute that it is a string. It is a user readable string that explains the purpose of a method, and is typically seen in the context menu or the Data Object Chooser.

**MethodParameters** is a class attribute that is a list of dictionaries. This is a collection of properties about the input data. Each element of the list corresponds to a single input dataset. The length of this list determines the number of input DataSets for the DataIterator (see section E.1.6) which the methodParameters will be passed to. Some common attributes for each dictionary include:

- **name** *(string)* A user-readable name for the input dataset specific to this method. Used to tag the DataSet when using it in a DataIterator.

- **validate** *(None or list of functions)* List of validation functions to check the dimensionality of the DataSet after the context is applied. For example, we could see if there are only two dimensions on the non-iterated dimensions, or if the shape of the non-iterated dimensions matches the shape of another DataSet. (when creating a method which requires both inputs, for example) These functions are stored in *dataview.utilities.validation*.

- **chunk** *(bool)* Whether or not the DataIterator will chunk over the dataset (e.g. multiple iterations at once)

- **edit** *(bool)* Whether the data object gets edited by the Method.

- **reshape** *(bool)* Whether the data object gets reshaped by the Method.

- **longname** *(str)* The long name of the data object, typically seen in a Data Object Chooser.

- **description** *(str)* Description of this data object, typically seen in a Data Object Chooser.

- **link** *(str)* ‘name’ string of another data object in this methodParameters, which share the same iterator as this one. Iterated dimensions must be the same. If there isn’t a link in this data object, this attribute shouldn’t be in this part of methodParameters.
One final class attribute is \textit{methodType}, seen in base classes for the type of Method. This simply tells the software whether classes which inherit this base type are Analyze, Process, or Display methods and sorts them accordingly in the menu system.

All Methods require two abstract routines that need to be filled. The first is \textit{create\_action\_from\_menu}, which creates a DVAction based on a menu choice. Several things can happen in this routine. In some cases the menu choice just flips some parameter (like a checked menu item). In that case `None` may be returned. In other cases the menu item will specifically describe what action needs to be taken and that action will be returned (note that the action should NOT be implemented at this point). Finally, in some cases user interaction may be required to determine the specifics of the action (e.g. menu commands ending in "..."). In that case the GUI should be presented to the user and the action should be fleshed out. It can then either be returned OR cancelled by returning `None`. It requires a QAction, the menu item which was called to call this routine, and a recipient, either a DataIterator or Viewer depending on the type of Method.

\textit{Execute} is the second abstract routine, which executes the method. This routine should contain the actual execution routine for the Method, regardless of where Method.execute() is called. It requires a DVAction, which specifies exactly how the Method is to be executed, and a recipient, which is typically a Datalterator or Viewer (the latter used in Display methods). The execute routine returns a boolean for whether the method has successfully run. All the actual use of Datalterator and analysis routines are to be located in this one routine: This is the workhorse of the Method class.

\textbf{E.8.2: Process Methods}

Process methods are Methods which only directly modify existing data. Since they modify data, they must all implement an undo capability. They have a base class named \textit{ProcessMethodBase} which all process methods must be subclassed from. Process methods may apply on one or more DataSets, but they never create new DataSets.

Process methods have a unique action – instead of using \textit{DVAction}, they use \textit{DVProcessAction}, as mentioned in section 5.5.2. They differ in that an undo action needs to be passed to it. If the method has an invertible function, this can be done by passing another \textit{DVProcessAction} whose Method is the inverse of this Method. For example, if one method multiples data, to undo it you can use a method which divides data.

Process methods have an additional abstract routine: \textit{finalize\_undo}. This routine is called after the method execution completes, if it is successful. It is the last chance to fill in the details about the undo before the action is placed in a History/Undo structure. It requires the DVProcessAction of the method, and the result code from the successful completion, which can be used to pass information to this routine. Typically, this routine will do nothing. If a file undo is
required, then it will be handled automatically by the action. If a functional undo is possible, then it can be either be created in the `create_action_from_menu` routine, in the execute command, or in the `finalize_undo` routine.

Examples of process methods include smoothing data, cropping data, applying background subtraction, and applying math on the data.

**E.8.3: Analyze Methods**

Analyze methods are Methods which create new data. They lack undo functionality since they do not modify already existing data. They have a base class named AnalyzeMethodBase, which all analyze methods are subclassed from. Analyze methods may require one or more DataSets and creates one or more DataSets.

All analyze methods create new DataSets by using the `create()` functionality in a DataIterator. As the DI.create() routine requires a dimset to pass through, a programmer can use DI.dimset(key) to grab the non-iterated dimensions of an input dimset to either use or modify for create().

After the output datasets have been successfully modified through the iteration process, the programmer can grab the output using `DI.output(key)` where key is the key of the output DataSet. The programmer typically will have to then create a DataSelector to view a slice of the data in a Viewer. Functions to help with this process are stored in `dataview.utilities.analyze`, which include functions to create dataselectors, create new viewers, and reshape arrays into certain formats needed for some analyze methods.

Examples of analyze methods include fourier transforms, autocorrelation, unsupervised machine learning algorithms like principal component analysis and clustering, and extracting statistics of the dataset.

**E.8.4: Display Methods**

Display methods do not act on DataSelectors or DataSets, but instead modify or create new Viewers. The user will use these methods to find new ways to view the datasets in the program.

The recipient for Display methods are viewers, rather than DataIterators. If the programmer wants to access the data stored in the viewer, they can grab the viewObject of the viewer. (e.g. `viewer.viewObject`)

Examples of display methods include setting the default context of the DataSelector stored in a viewer, creating a histogram of the data, viewing a region of interest or pan/zoom toolbar, and displaying a series of one dimensional curves from an image viewer.
E.9 Summary

Appendix F of this thesis contains a number of examples of working scripts that are a part of each module. Specifically, it contains a file handler, example Process, Analyze, and Display methods, and two example Viewers, one based on matplotlib, and the other based on a Qt Widget, as well as an example LocatorWidget. The Methods and FileHandlers show how to create logs in the program, useful as part of the macro system of the software.
Appendix F

Example DataView Module Code

This appendix gives example code of each module. It includes a file handler, process, analyze, and display methods, a Matplotlib viewer, a Qt viewer, and a locator widget.

F.1: Example FileHandler: FilePNG

```python
# -*- coding: utf-8 -*-

.. py:module:: dataview.filehandlers.filepng

FilePNG Class

File Handling for png files

:Version: 1
:Author: Eric Hudson
:Date: June 24, 2015

import dataview.data.datasets as dvd
import dataview.data.locate as dvlocate
import dataview.data.dimensions as dvdim
import dataview.data.dataselector as dvdatsel
from dataview.main.dvlog import get_logger
from dataview.data.dvcollection import DVCollection
from dataview.filehandlers.filehandlerbase import FileHandlerBase
from dataview.viewers.viewgroup import ViewGroup as VG
from dataview.main.dvunits import ureg
from skimage import io

rootlog = get_logger('root')

class FilePNG(FileHandlerBase):
    
    This is the file handler for TEMPLATE

    Attributes
    ----------
    info : dict
        A dictionary with the following entries:
        version : float
            The version number. Override in implementation subclasses
        whatsThis : str
            A (user readable) description of the type of data this handler handles
        extensions : set of str
```
The extensions (e.g. jpg, jpeg, sxm, ...) handled by this filehandler
Note that some extensions (e.g. 'dat') might be handled by multiple
handlers, in which case the user will be queried as to which one
to try

filters : set of str
A set of user defined categories into which we might sort the type
of file handled by this handler. For example, jpg, tif, etc might
be {"Images"} while 'sxm' might be {"STM Files","Topographies"}

needConfiguration : bool
Do we need extra configuration information to save the file?
If so it is assumed that there is a configuration gui in the
filehandler to allow setting of these extra parameters
(e.g. jpg needs a "Quality" set)

handledTypes : set of str
A set of types of data that can be written to

canRead : bool
Can we handle reading this kind of file?

Methods
-------
load(filename)
Load a single file (path `filename`)

save(filetype, obj, filename, configuration)
Writes `object` of type `filetype` to `filename`
using `configuration` information

get_configuration(default)
If necessary, generates a "configuration" dict with additional
information about how to save data (e.g. "Quality" in a jpg file)
This is typically done through a GUI. The "default" configuration
should be used for initialization, if present.
If not necessary, just "pass"

```
def __init__(self, *args, **kwargs):
    pass
# FileHandler classes should never need to be instantiated
```

FILETYPES = {
    '1D', # 1D datasets
    '2D', # 2D datasets
    'data', # any dimension dataset (if listing this, don't need 1D, 2D)
    'viewer', # the display (either as image or to reconstruct)
    'dataGroup', # a group of related data
    'viewerGroup', # a group of viewers (including notes)
    'session', # the entire session (all current data & viewers)
    'note', # text like notes, logs, ...
    'macro' # a list of actions (for automating processing)
}

info = {
    'version': 1.0,
    'whatsThis': 'Portable Network Graphics is a raster graphics ' +
    'file format that supports lossless data compression.',
    'extensions': {'png'},
    'filters': {'Images'},
'needConfiguration': False,
'handledTypes': {'2D', 'viewer'},
'canRead': True
}

@classmethod
def load(cls, filename):
    ""
    Load the single file referred to by `filename`. That a file exists
    with this filename will already be vetted by the top level filehandler
    If `canRead` is false, this should just be pass
    ""
    Parameters
    ----------
    filename : str
        a single, existing file to be loaded
    Returns
    -------
    collection : dataview.data.datasets.DVCollection
        A collection of datagroups and datasets
    # Extracts the name of the DataSet from the filename
dataname = filename.split(\'\')[\-1].split(\'\')[0]

    # Read the image file into a numpy array
    alldata = io.imread(filename)
    if len(alldata.shape) == 3:
        x, y, colors = alldata.shape
        channels = [\'r\', \'g\', \'b\', \'alpha\'][0:colors]
    else:
        x, y = alldata.shape
        channels = [\'value\']
    # Swap axes into DataView format
    alldata = alldata.swapaxes(0, 2)  # (x, y, v) -> (v, y, x)
    rootlog.info("Data Shape: {}".format(alldata.shape))

    # Set up dvdim and the DimSet
    xdim = dvdim.Dimension(name=\'x\', numElements=x, convert=('Null', None, (None, None)), unit=ureg.dimensionless)  #axis=2
    ydim = dvdim.Dimension(name=\'y\', numElements=y, convert=('Null', None, (None, None)), unit=ureg.dimensionless)  #axis=1
    vals = dvdim.DTDimension(channels, units=[ureg.dimensionless] * len(alldata.shape),
                           dtype=alldata.dtype)  #axis=0
    ds = dvdim.DimSet([vals, ydim, xdim])
    # Create Dataset
dset = dvd.DataSet(ds, name=dataname, temp=False)
dset.array = alldata
    # Bind Selectors
dset.bind_selector("Image", [\'Pcolor.0\', None, None])
    # Create collection
collection = DVCollection(name=dataname + \' Collection\')
collection.append(dset, name=dataname)
    return collection

@classmethod
def display(cls, filename, collection):
    "

Displays a default number of relevant viewers: At least a TreeViewer for the collection returned from the loading process.

Parameters
----------
filename : str
    a single, existing file to be loaded
collection : dataview.data.datasets.DVCollection
    A collection of datagroups and datasets

# Extract datagroups and/or datasets with bound selectors
datename = filename.split(‘\’)[-1].split(‘.’)[0]
dset = collection[dataname]

# Create viewers
# TODO: Fix viewers
image = dset.bound_selectors["Image"]
viewgroup = VG(title=filename)
plotV = viewgroup.add_viewer("ImgViewer", image)
# Add Combobox
plotV.addLWidget(’ComboBox’, image.get_locator(’color’), key=’comboboxes’)

# Add Cursor
plot_xy = dvlocate.Picker(valueArray=[image.dimset[1], image.dimset[0]])
# Create picker
plotV.addLWidget(’MPLCursor’, plot_xy, key=’cursor’)
plotV.display()

# Create tree viewer
tree = viewgroup.add_viewer("TreeViewer", collection)
return

@classmethod
def load_display(cls, filename):
    Loads a file into a collection and displays a default number of relevant viewers. Method applied when the file is loaded in the main menu.
    
    collection = cls.load(filename)
    cls.display(filename, collection)
    return

@classmethod
def save(cls, filetype, obj, filename, configuration=None):
    Writes `object` of type `filetype` to `filename` using `configuration` information
    
    Parameters
    ----------
    filetype : str
        One of the FHB.filetypes (e.g. 'data' or 'viewer')
    obj : various
        The object to save to the file. The type and how it is handled depends on filetype
    filename : str
        The filename to write to. It has been vetted as a valid path and overwrite is allowed if it already exists
    configuration : dict
        A dictionary of configuration information
This is class dependent

if configuration is None:
    configuration = {}
# assumes data is a data object
if isinstance(obj, dvdatsel.DataSelector):
    # grabs parent object
    # TODO: We might want to add functionality to grab slices of data
    # that fit the criteria
    array = obj.parent.datablock.array
elif isinstance(obj, dvd.DataSet):
    array = obj.datablock.array
elif isinstance(obj, dvd.DataBlock):
    array = obj.array
else:
    # no other object type can be set
    return
# array must be three dimensions
if array.ndim != 3:
    rootlog.error('FileHandler {}: {} not saved to {} (# array
dimensions: {}, needs to be 3)'.format(cls, obj, filename, array.ndim))
    return
# third dimension must be channel dimension: length of 1 or 3
# DataView typically places channels first with inversed dimensions
if array.shape[0] in (1, 3):
    array = array.swapaxes(2, 1)
    array = array.swapaxes(2, 0)  # (x, y, v) -> (v, y, x)
# this catches situations when the channel dimension is not correct
elif array.shape[-1] not in (1, 3):
    rootlog.error('FileHandler {}: {} not saved to {} (Channels: {},
needs to be 1 or 3)'.format(cls, obj, filename, array.shape[-1]))
    return
# save image
try:
    io.imsave(filename, array)
except Exception as e:
    rootlog.error('FileHandler {}: {} not saved to {}: {}'.format(cls, obj, filename, e))
    rootlog.info('FileHandler {}: Save {} (of type {}) to file {} using
configuration {}'.format(cls, obj, filetype, filename, configuration))
    return

@classmethod
def get_configuration(cls, default={}):
    ""
    Generates a "configuration" dict with additional information about how
to save data (e.g. "Quality" in a jpg file)
    ""
    The main file handler will maintain a list of all filehandlers and
    the information (data & viewers) they have saved indexing the last
    save configuration. This will be used if the same information is saved
    again, unless the user specifically requests to change the
    configuration. In that case, or in the case where information has
    not yet been saved in this format, this method will be called
    to generate the configuration (typically via a gui)
Parameters
---------
default : dict

A "default" dict to use in initializing parameters in the gui, for example. This will typically be the configuration used in the last save.

Returns
-------
dict

Configuration information (to be used when writing to a file)

configuration = default

return configuration
F.2: Example Process Method: GaussFilter

```python
#!/usr/bin/env python3
#
# coding: utf-8
#

.. py:module:: dataview.methods.process.gaussfilter

GaussFilter Class

Smooths the data using a gaussian filter

.. version:: 1
.. author: Bill Dusch and Joseph McDoal
.. date: March 25, 2016
.. update: May 25, 2018; ewh; Modified to use new DataIterators & DVAction

from PyQt5 import QtWidgets
from dataview.methods.process.processbase import ProcessMethodBase
from dataview.main.dvaction import DVProcessAction
from dataview.main.dvlog import get_logger
from scipy.ndimage.filters import gaussian_filter

rootlog = get_logger('root')

class GaussFilter(ProcessMethodBase):
    
    A Gaussian filter is a filter whose impulse response is a Gaussian function.
The filter is characterized by its standard deviation, which determines the
width of the smoothing.

    Attributes
    ----------
    info : dict
        This is one of two attributes which must be edited in each class
    that inherits from the MethodBase base class.
    A dictionary with the following entries:
    version : float
        The version number. This is important as it is used to determine
        whether menus need to be regenerated and whether action lists can
        be run in the same fashion (ie for automated Methods). Override
        in implementation subclasses
    submenu : string
        Used for grouping together methods in a single submenu,
        this can either be '', in which case the menuitem(s)
        will be in the main part of the relevant menu (e.g. under
        Process or Analyze depending on the `MethodType`) or
        a menu name (e.g. `Special`) which will create that submenu
        and put this and any other methods which list the same
        submenu in it, or a submenu structure (e.g. `Special.2015`) in
        which case a nested submenu structure will be created
    menus : dict
        A collection of menus, keyed by the `vista` (see `VISTAS`) in which
        that menu is to be used. The menu format is flexible: see
```
Menus (for DVMenu) can be defined by simple text, by a dict with any subset of the following keys (you can abbreviate them):

- 'text', 'icon', 'shortcut', 'tip', 'checked', 'whatsThis', 'name'
- or with lists of multiple items or nested lists to make submenus.

Note that with lists that the first item is the menu name and the other items show up as a sublist.

Examples

```python
{ '1D': 'Line subtraction...' }  # a simple string menu item
{ '1D': {'te': 'Line subtraction...', 'sh': 'Ctrl+L'} }  # dict defined
{ '2D': ['Background Subtraction', 'Plane', '2nd order'] }
{ '1D': 'Line sub', '2D': 'Plane sub' }
```

methodParameters: List of dicts

A collection of properties about the input data. Each element of the list corresponds to a single input dataset. The length of this list determines the number of input DataSets for the DataIterator.

Some common attributes for each dictionary:

- **name**: (str) User-readable name for the input data specific to this method.
- **validate**: (None or list of functions): List of validation functions to check the dimensionality of the DataSet after the context is applied.
- **chunk**: (bool) Whether or not the DataIterator will chunk over this dataset (e.g. multiple iterations at once)
- **edit**: (bool) Whether the data object gets edited by the Method.
- **reshape**: (bool) Whether the data object gets reshaped by the Method.
- **longname**: (str) The long name of the data object, typically seen in a Data Object Chooser.
- **description**: (str) Description of the method.
- **link**: (str) 'name' string of another data object in methodParameters which share the same iteration as this one. (Iterated dimensions must be the same.)

If there isn't a link, this attribute shouldn't be in this part of the methodParameters.

MethodType : str
The 'type' of Method this is (e.g. Process, Analyze). This should be overridden in abstract subclasses but probably not touched by implementation classes.

Methods

```python
execute(action, DI)
Execute the Method on data pointed to by the `DataIterator`, with details in `action` (eg with the crop dvdim and given dataiterator)
```

create_action_from_menu(menuitem, menu, dataiterator)

Creates an action given a selected `menuitem` (and the `menu` from which it was selected) and the `dataiterator` of the menu call

```python
""
```

```python
info = {
    'version': 1.0,  # Update version when you make substantive changes
    'submenu': 'Smooth'  # Optional: allows grouping of some methods in submenu
}
```

userExplanation = ('Apply a gaussian filter to smooth the data')

# Information about the input datasets
methodParameters = [
    {
        'name': 'in',  # short name for this dataset
        'validate': None,  # No limit to viewed dimensions
        'chunk': True,  # whether or not we are chunking in the iterator
        'edit': True,  # Boolean - does this dataobject get edited by the method
        'reshape': False,  # Boolean - does this dataobject get reshaped by the method?
        'longname': 'Data',  # Long name of the dataset
        'description': 'Dataset to smooth'  # description of dataset
    }
]

menus = {'1D': {'te': '1D Gaussian filter...', 'sh': 'Ctrl+G'},
         '2D': {'te': '2D Gaussian filter...', 'sh': 'Ctrl+G'},
         }

@classmethod
def execute(cls, action, DI):
    ""
    Execute the Method (this should be the final execution method for
    the Method regardless of entry point)
    ""
    Parameters
    ----------
    action : DVAction
        'action' specifies exactly how the Method is to be executed
    DI : dataview.data.DataIterator
        Indicates how to iterate over the data on which the Method is to be executed
    Returns
    -------
    bool
        Was the execution successfully completed?
    ""
    sigmarray = [0] + [action.details['sigma']] * len(DI.dimset())
    for array in DI:
        print("Pause")
        DI.update(gaussian_filter(array, sigmarray))
    return True

@classmethod
def create_action_from_menu(cls, menuitem, DI):
    ""
    The GaussFilter create_action_from_menu creates a dialog box which allows
    the user to input a value for sigma, and if the value is good (numeric)
    it applies the sigma value to the details dictionary of an action. No special
    undo is created (undo=None) because GaussFilter is not invertable, so undo will
    load an undo file.
    Parameters
    ----------
    menuitem : QAction
        The menu item which was called. menuitem.text() is the text,
        additional info may be in menuitem.data(), a dict which contains
        at least "menu," the sub-QMenu in which this QAction exists
    DI : DataIterator
        The object used to iterate over a dataset, containing information
about the context of the menu. This will include the dimensionality of
the data (and info about how to make it from the data selector) as well
as info about how it was called (e.g. from a display, or data list...)

Returns
-------
DVAction

The action to be performed based on the menu call (or None)

```python
dlog = FilterSigma()
DI.usechunks = True
if dlog.exec():
    sigma = dlog.sigma()
    action = DVProcessAction(method=cls, description='GaussFilter
 sigma={}'.format(sigma),
                     details={'sigma': sigma}, undo=None, undoRecipient=DI)
    return action
else:
    return None
```

@classmethod
def finalize_undo(cls, action, result):
    ""
    No processing required for gaussian filter, but we must override
    the routine because it was declared abstract
    ""
    pass

class FilterSigma(QtWidgets.QDialog):
    """Dialog box to get from the user the standard deviation for a Gaussian Filter."""
    def __init__(self):
        QtWidgets.QDialog.__init__(self)
        self.setup()
    def setup(self):
        ""
        Sets up the dialog box: Simple form with text:QlineEdit with OK
        and Cancel buttons
        ""
        self.setWindowTitle('Gaussian Filter')
        self.label = QtWidgets.QLabel('Sigma:')
        self.edit = QtWidgets.QLineEdit()
        self.edit.setFixedWidth(35)
        self.form = QtWidgets.QFormLayout()
        self.form.addRow(self.label, self.edit)
        self.ok = QtWidgets.QPushButton('OK')
        self.cancel = QtWidgets.QPushButton('Cancel')
        self.form.addRow(self.ok, self.cancel)
        self.form.setForm(self.form)
        self.ok.clicked.connect(self.connect_ok)
        self.cancel.clicked.connect(self.reject)
    def connect_ok(self):
        ""
        Check to see if the sigma value is valid; must be a positive float
        ""
        try:
            sigma = float(self.edit.text())
except:
    text = 'Sigma must be a positive float'
    QtWidgets.QMessageBox.warning(None, 'Error', text)
else:
    if sigma >= 0:
        self.accept()
    else:
        text = 'Sigma must be a positive float'
        QtWidgets.QMessageBox.warning(None, 'Error', text)

def sigma(self):
    """Returns the value of sigma from the QLineEdit for create_action_from_menu to apply to the action""

    return float(self.edit.text())
The FFT may be either one or two dimensional; dimensionality depends on the context.

As the FFT is a complex number, the user can choose from extracting the Amplitude, Phase,
Real or Imaginary components of the FFT.

Attributes

Attributes

info : dict

This is one of two attributes which must be edited in each class
that inherits from the MethodBase base class.
A dictionary with the following entries:

version : float

The version number. This is important as it is used to determine
whether menus need to be regenerated and whether action lists can
be run in the same fashion (ie for automated Methods). Override
in implementation subclasses

submenu : string

Used for grouping together methods in a single submenu,
this can either be '', in which case the menuItem(s)
will be in the main part of the relevant menu (e.g. under
Process or Analyze depending on the 'MethodType') or
a menu name (e.g. 'Special') which will create that submenu
and put this and any other methods which list the same
submenu in it, or a submenu structure (e.g. 'Special.2015') in
which case a nested submenu structure will be created

menus : dict

A collection of menus, keyed by the 'vista' (see 'VISTAS') in which
that menu is to be used. The menu format is flexible: see
DVMenu.add_item for details.

Menus (for DVMenu) can be defined by simple text, by a dict
with any subset of the following keys (you can abbreviate them):
['text','icon','shortcut','tip','checked','whatsThis','name']
or with lists of multiple items or nested lists to make submenus.
Note that with lists that the first item is the menu name and the
other items show up as a sublist

Examples

--------

{ '1D' : 'Line subtraction...' } # a simple string menu item
{ '1D' : {'te':'Line subtraction...','sh':'Ctrl+L'} } # dict defined
{ '2D' : ['Background Subtraction','Plane','2nd order'] }
{ '1D' : 'Line sub', '2D' : 'Plane sub' }

methodParameters: List of dicts

A collection of properties about the input data. Each element of the list corresponds to
a single input dataset. The length of this list determines the number of input DataSets
for the DataIterator.

Some common attributes for each dictionary:

name: (str) User-readable name for the input data specific to this method.
validate: (None or list of functions): List of validation functions to check the
dimensionality of the DataSet after the context is applied.
chunk: (bool) Whether or not the DataIterator will chunk over this dataset (e.g.
multiple iterations at once)
edit: (bool) Whether the data object gets edited by the Method.
reshape: (bool) Whether the data object gets reshaped by the Method.
longname: (str) The long name of the data object, typically seen in a Data Object
Chooser.
description

link: (str) 'name' string of another data object in methodParameters which share the
same iteration as this one. (Iterated dimensions must be the same.)
If there isn't a link, this attribute shouldn't be in this part of the methodParameters.

MethodType : str
The 'type' of Method this is (e.g. Process, Analyze). This should be overridden in abstract subclasses but probably not touched by implementation classes

Methods
-------
execute(action, DI)
Execute the Method on data pointed to by the `DataIterator`, with details in `action` (eg with the crop dvdim and given dataiterator)

create_action_from_menu(menuitem, menu, DI)
Creates an action given a selected `menuitem` (and the `menu` from which it was selected) and the `DataIterator` of the menu call

```
info = {
    'version': 1.0,  # Update version when you make substantive changes
    'submenu': '',   # Optional: allows grouping of some methods in submenu

    }
userExplanation = 'Perform a Fast Fourier Transform'

# Information about the input datasets
methodParameters = [
    {
        'name': 'in',  # short name for this dataset
        'validate': [check_dims((range(1, 3)))]),  # one or two dimensional FFTs
        'chunk': True,  # whether or not we are chunking in the iterator
        'edit': False,  # Boolean - does this dataobject get edited by the method
        'reshape': False,  # Boolean - does this dataobject get reshaped by the method?
        'longname': 'Signal',  # Long name of the dataset
        'description': 'Dataset to apply the fourier transform on. FFT applied on viewed dimensions'  # description of dataset
    }
]

menus = {'1D': ['FFT', 'Amplitude', 'Phase', 'Real', 'Imaginary'],
    '2D': ['FFT', 'Amplitude', 'Phase', 'Real', 'Imaginary']}

@classmethod
def execute(cls, action, DI):
    ""
    Execute the Method (this should be the final execution method for the Method regardless of entry point)

    Parameters
    ----------
    action : DVAction
        `action` specifies exactly how the Method is to be executed as well as the dataSelector dataiterator
    DI : dataview.data.DataIterator
        Indicates how to iterate over the data on which the Method is to be executed

    Returns
    -------
    bool
Was the execution successfully completed?

rootlog.info('Executing Method %s with action %s on ds %s' % (cls.__name__, action, DI.dataset_parameters[0]['data']))

# Setting up frequencies and units

# Create new dimensionset - same shape as original, different units (frequency space)
orig_dimset = DI.dimset('in') # does NOT include iterated dimensions
iter_dims = DI.iter_dims('in')
fft_dimset = orig_dimset.copy()
dilen = len(fft_dimset)

# change datatype units if angle
if action.details['FFT'] == 'Phase':
    fft_dimset.dataDim.units = [1 * ureg.degree] * len(fft_dimset.dataDim)

# make a loop for dilin
lens, axes = [], []
for i, dim in enumerate(fft_dimset):
    lens.append(len(dim))
    fft_dimset[dim.name] = cls.frequency(dim)
    axis = DI.iter_index(dim.name, 'in')
    axes.append(axis)
mullens = reduce(lambda x, y: x*y, lens)

# Create new DataSet
DI.create('fft', fft_dimset, 'in', view=True)

# Iterate!
for array in DI:
    DI.update(fftshift(transform(fft_func(array, axis=axes) / mullens)), name='fft')

# For reference, here is the created output FFT dataset
fft_dset = DI.output('fft')

# View datasets
if action.details['FFT'] == 'Amplitude':
    parameters = {'norm': 'log', 'fourier': True}
else:
    parameters = {'norm': None, 'fourier': True}
for viewObj in DI.to_view:
    VB.default_viewer(viewObj, parameters=parameters)
return True

@classmethod
def frequency(cls, dim):
    ""

120
Convert a dimension to frequency space.

Parameters

- `dim`: dimension in spatial or temporal coordinates

Returns

- `dim`: dimension in frequency space

```python
# start by copying the dimension
fdim = copy(dim)
# get spacing
dx = dim.getv(1) - dim.getv(0)
# invert unit
fdim.unit = 1.0 / fdim.unit
# this needs to be converted to new format...
fdim.linspace(-np.pi / dx, np.pi / dx, len(fdim))
return fdim
```

@classmethod
def angle(cls, array):
    # This is mostly for formatting purposes - convert to degrees
    array = np.angle(array, deg=True)
    return array

@classmethod
def create_action_from_menu(cls, menuitem, DI):
    """
    Creates an `action` based on a menu choice
    
    Several things can happen here. In some cases the menu choice just
    flips some parameter (like a checked menu item). In that case `None`
    may be returned. In other cases the menu item will specifically
    describe what action needs to be taken and that action will be returned
    (note that the action should NOT be implemented at this point).
    Finally, in some cases user interaction may be required to determine
    the specifics of the action (e.g. menu commands ending in "..."). In
    that case the gui should be presented to the user and the action
    should be fleshed out. It can then either be returned OR cancelled
    by returning `None`.
    """

    Parameters

    - `menuitem`: QAction
      The menu item which was called. `menuitem.text()` is the text,
      additional info may be in `menuitem.data()`, a dict which contains
      at least "menu," the sub-QMenu in which this QAction exists
    
    - `DI`: DataIterator
      The object used to iterate over a dataset, containing information
      about the dataiterator of the menu. This will include the dimensionality of
      the data (and info about how to make it from the data selector) as well
      as info about how it was called (e.g. from a display, or data list...)

    Returns

    - `DVAction`
      The action to be performed based on the menu call (or None)

    """
    rootlog.info('{}::create_action_from_menu, menuitem = {}'.
        format(cls.__name__, menuitem))
```
FFT = menuitem.text()

# Probably should include which axis to do FFT over
# Need to check if right dataiterator
action = DVAction(method=cls, description='FFT ({})'.format(FFT),
                  details={'FFT': FFT})
return action

F.4: Example Display Method: Histogram

This Display method displays a histogram from the current DataSelector in the
viewer - same locators but views the distribution of points.

Attributes
----------
info : dict
This is one of two attributes which must be edited in each class
that inherits from the MethodBase base class.
A dictionary with the following entries:
version : float
    The version number. This is important as it is used to determine
    whether menus need to be regenerated and whether action lists can
    be run in the same fashion (i.e., for automated Methods). Override
    in implementation subclasses
submenu : string
    Used for grouping together methods in a single submenu,
    this can either be '', in which case the menuitem(s)
    will be in the main part of the relevant menu (e.g., under
    Process or Analyze depending on the 'MethodType') or
    a menu name (e.g., 'Special') which will create that submenu
    and put this and any other methods which list the same
submenu in it, or a submenu structure (e.g. 'Special.2015') in which case a nested submenu structure will be created.

menus : dict
A collection of menus, keyed by the `vista` (see `VISTAS`) in which that menu is to be used. The menu format is flexible: see DVMenu.add_item for details.

Menus (for DVMenu) can be defined by simple text, by a dict with any subset of the following keys (you can abbreviate them):

- `text`
- `icon`
- `shortcut`
- `tip`
- `checked`
- `whatsThis`
- `name`

or with lists of multiple items or nested lists to make submenus.

Note that with lists that the first item is the menu name and the other items show up as a sublist.

Examples
--------

{ '1D' : 'Line subtraction...' } # a simple string menu item
{ '1D' : {'te':'Line subtraction...','sh':'Ctrl+L'} } # dict defined
{ '2D' : ['Background Subtraction','Plane','2nd order'] }
{ '1D' : 'Line sub', '2D' : 'Plane sub' }

Methods
-------

execute(action, viewer)
Execute the Method on the viewer, with details in `action` (eg with the crop dvdim and given dataiterator)

create_action_from_menu(menuitem, menu, dataiterator)
Creates an action given a selected `menuitem` (and the `menu` from which it was selected) and the `dataiterator` of the menu call

info = {
    'version': 1.0, # Update version when you make substantive changes
    'submenu': ''  # Optional: allows grouping of some methods in submenu}

userExplanation = ('Open up a histogram of data based on the distribution of data ' +
                   'in the current display')

menus = {'1D': 'Histogram',
         '2D': 'Histogram'}

# Here are the vistas which should be considered in `menus`
# VISTAS = [ # The different types
#     '1D', # 1 dvdim data (viewer if display method) or collection thereof
#     '2D', # 2 dimensional
#     '3D', # 3 dimensional
#     'ND', # >3 dimensional
#     'palette' # palette editor
# ]

@classmethod
def execute(cls, action, viewer):
    ""
    Execute the Display (this should be the final execution method for the Method regardless of entry point)
Parameters

--------

action : DVAction

'`action` specifies exactly how the Method is to be executed as well as the dataSelector dataiterator

viewer : dataview.viewers.ViewerBase

Indicates the viewer on which the Method is to be executed

Returns

-------

bool

Was the execution successfully completed?

```python
rootlog.info('Executing Method %s with action %s on viewer %s' %
             (cls.__name__, action, viewer))
```

# Here's the question: Does an identical dataselector need to be copied or can we use the old one?

```python
DS = viewer.viewObject
title = DS.name + ' (Histogram)'
new_viewer = cls.setup_hist_viewer(viewer, DS, parameters={'fourier': False, 'autoscale': True, 'title': title})
return True
```

```python
@classmethod
def create_hist_dataselector(cls, dataset, oldDS, name=' '):
    ""
    Creates a new DataSelector from an originally existing dataset
    Parameters
    ----------
    dataset
    oldDS
    name
    ""
    newDS = DataSelector(name, dataset)
    locator_list = []
    for task in oldDS:
        if isinstance(task, dst.DSTaskLocator):
            # these should be the same
            destroy = task.parameters['destroy']
            create = task.parameters['create']
            # grab the old locator - we need to create a new one
            old_locator = task.parameters['locator']
            dimList = old_locator._dimList
            locator = old_locator
            if len(dimList) > 0:
                parameters = {'locator': locator, 'create': create, 'destroy': destroy}
                new_task = dst.DSTaskLocator(parameters=parameters)
                newDS.append(new_task)
                locator_list.append(locator)
        elif not isinstance(task, dst.DSTaskLocatorHandler):
            newDS.append(copy(task))
        # And add the LocatorHandler
        if len(newDS) > 0:
            newDS.append(dst.DSTaskLocatorHandler(parameters={'locators':
                locator_list}))
    newDS.process()
```
return newDS

@classmethod
def setup_hist_viewer(cls, orig_viewer, newDS, short=15, parameters=None):
    
    Helper method to set up the viewer in an Analyze method.
    Parameters
    ----------
    action: DVAction of the method, which stores viewgroup information
    newDS: New DataSelector that the Analyze method created.
    short: If one dimensional dataset, length of dimension for threshold to view as
    a Table instead of a Plot.
    Returns
    -------
    viewer: Viewer
    The viewer that is displayed.
    
    parameters = {} if parameters is None else parameters
    # Create viewer
    new_viewer = add_viewer(orig_viewer.viewGroup)("HistViewer", newDS, parameters=parameters)
    # Set up locatorwidgets
    comboloc = [task.parameters['locator'].name for task in newDS if isinstance(task, dst.DSTaskLocator) and len(task.parameters['locator']._dimList) == 1]
    for name in comboloc:
        new_viewer.addLWidget('ComboBox', newDS.get_locator(name), key='comboboxes')
    new_viewer.display()
    return new_viewer

@classmethod
def create_action_from_menu(cls, menuitem, viewer):
    
    Creates an 'action' based on a menu choice
    Parameters
    ----------
    menuitem : QAction
    The menu item which was called. menuitem.text() is the text,
    additional info may be in menuitem.data(), a dict which contains
    at least "menu," the sub-QMenu in which this QAction exists
    dataiterator : DataIterator
    The object used to iterate over a dataset, containing information
    about the dataiterator of the menu. This will include the dimensionality of
    the data (and info about how to make it from the data selector) as well
    as info about how it was called (e.g. from a display, or data list...)
    Returns
The action to be performed based on the menu call (or None)

```python
action = DVAction(method=cls, description='Histogram',
                  details={})
```

```python
print('{}::create_action_from_menu, menuitem = 
      {}\'.format(cls.__name__, menuitem.text()))
```

F.5: Example Matplotlib Viewer: ImgViewer

```python
.. py:module:: dataview.viewers.mplviewers.imgviewer

=======================================
ImgViewer Class
=======================================

The ImgViewer uses matplotlib imshow to display a 2D image

```

```python
from dataview.viewers.mplviewers.mplviewerbase import MPLViewerBase, DVCursor
from matplotlib.colors import LogNorm
from dataview.main.dvunits import ureg
import dataview.data.dataselector

class ImgViewer(MPLViewerBase):
    """
    Parameters
    ----------
    viewObject : object
      The object to be viewed in this viewer. Note that it is assumed
      that this object has already been checked as "allowed" (obeying
      the viewclass and constraints). Defaults to none, in which case
      an empty viewer will be opened

    Attributes
    ----------
    context : None or data.context
      The context applied to the viewObject.

    info : dict
      A dictionary with the following entries:
      version : float
        The version number. This is important as it is used to determine
        whether menus need to be regenerated and whether action lists can
        be run in the same fashion (ie for automated Methods). Override
        in implementation subclasses
```
viewclass : class
Each viewer type can display only one class of object. For
example, imageviewers and plots will view DataSelectors,
while a property viewer might view a viewer. In the case that
the viewer should be able to view more than one kind of object
then superclass the two objects and view that (e.g. a list of
data objects and a list of dataSelector objects might both
be viewed by the same viewer, so make them subclasses of
an objList obj)

constraints : dict of method : value
To see if a specific viewclass object is suitable for viewing
in this viewer, each of the method keys in this dict will
be called and checked against the given values. The viewer
will only work if the values match. For example, a viewer
might only work on 2D data, so {'numDimensions':2}.

displaymenu : dvmenu descriptor
The top half of the "display" menu -- methods specific to this
viewer (the bottom half are Display Methods from DisplayMethodBase).

Menus (for DVMenu) can be defined by simple text, by a dict
with any subset of the following keys (you can abbreviate them):
['text','icon','shortcut','tip','checked','whatsThis','name']
or with lists of multiple items or nested lists to make submenus

Examples
--------
'Show error bars...' # a simple string menu item
{'te':'Show error bars...','sh':'Ctrl+B'} # dict defined

Methods
-------

```
info = {
    'version': 1.0,
    'viewclass': dataview.data.dataselector.DataSelector,
    'constraints': [{ndim': 2}, {ndim': 3, 'shape[2]': {3, 4}}],
    'viewvista': "2D" # The default vista for data in this viewer
}
```
displaymenu = {} # The portion of the display menu specific to this viewer
def __init__(self, *args, parameters={}, **kwargs):
    self.name = "ImgViewer"
    self.parameters = parameters
    self.norm = parameters['norm'] if 'norm' in parameters else None
    self.fourier = parameters['fourier'] if 'fourier' in parameters else False
    self.autoscale = parameters['autoscale'] if 'autoscale' in parameters else False
    MPLViewerBase.__init__(self, *args, parameters=parameters, **kwargs)
    self.setup()

def setup(self):
    
    Sets up the UI.
    
    MPLViewerBase.setup(self)
    # Create plot
    self.plot()
    # Replot when dataset is changed by a method
    self.viewObject.history.connect(self.replot)
# Refresh when dataselector pickers change
self.viewObject.connect_process(self.refresh)

def plot(self):
    """
    Plots the image for the first time
    """
    data = self.get_data()
    self.image = self.canvas.axes.imshow(data, origin="lower")
    self.label()
    self.adjust_limits(True)
    self.canvas.axes.set_aspect('auto')
    self.colorbar = self.canvas.fig.colorbar(self.image, ax=self.canvas.axes)
    self.canvas.draw()

def replot(self, info):
    """
    Replots the image, typically when a method is applied
    Parameters
    ----------
    info: emitter signal from the DataSelector, a list of booleans.
    0: DataSelector has updated a DTDimension
    """
    self.label()
    self.adjust_limits(True)
    self.refresh(info)

def refresh(self, info):
    """
    Repaints the image.
    Parameters
    ----------
    info: emitter signal from the DataSelector, a list of booleans.
    0: DataSelector has updated a DTDimension
    """
    data = self.get_data()
    self.image.set_data(data)
    if self.norm == 'log':
        self.image.set_norm(LogNorm(vmin=self.min, vmax=self.max))
    else:
        self.image.set_clim([self.min, self.max])
    if self.autoscale or info[0]:
        self.label()
        # set this to true or false to adjust color limits...
        if info[0]:
            self.adjust_limits(norm=True)
        else:
            self.adjust_limits(norm=False)
    self.canvas.axes.set_aspect('auto')
    self.canvas.draw()

def adjust_limits(self, norm=True):
    """
    Adjust the limits of the plot.
    Parameters
    ----------
    norm: Normalize the image, resetting the minimum and maximum of the Z-scale.
    (This is non-instantaneous.)
    """
    Returns
    -------
.. py:module:: dataview.viewers.treeviewer

F.6: Example Qt Viewer: TreeViewer

---

1. """
2. .. py:module:: dataview.viewers.treeviewer
3. ************************************************************************
4. TreeViewer Class
5. ************************************************************************
6. The TreeViewer is a Tree Widget window which displays a tree diagram
7. of a Data Collection - for example, the collection of data opened from
8. a file.
9. """
10. """
11. :Version: 1
12. :Author: Bill Dusch
13. :Date: May 5, 2016
14. 
15. from PyQt5 import QtWidgets, QtCore
16. from dataview.data.datasets import DataSet
17. from dataview.data.dvcollection import DVCollection
18. from dataview.viewers.viewerbase import ViewerBase
19. import dataview.data.datasets
20. from dataview.main.dvlog import get_logger
21. 
22. rootlog = get_logger('root')
23. 
24. class TreeViewer(ViewerBase):
25.     ""
26.     Parameters
27.     ----------
28.     viewObject : object
29.         The object to be viewed in this viewer. Note that it is assumed
30.         that this object has already been checked as "allowed" (obeying
31.         the viewclass and constraints). Defaults to none, in which case
32.         an empty viewer will be opened
33.     Attributes
34.     ----------
35.     viewWidget: PyQt QWidget
36.         The QWidget to be set in the Viewer.
37.     locatorwidgets: dict
38.         Dictionary holding LocatorWidgets used by the Viewer.
39.     dataiterator : None or data.dataiterator.DataIterator
40.         The dataiterator applied to the viewObject.
41.     info : dict
42.         A dictionary with the following entries:
43.         version : float
44.             The version number. This is important as it is used to determine
45.             whether menus need to be regenerated and whether action lists can
46.             be run in the same fashion (ie for automated Methods). Override
47.             in implementation subclasses
48.         viewclass : class
49.             Each viewer type can display only one class of object. For
50.             example, imageviewers and plots will view DataSelectors,
51.             while a property viewer might view a viewer. In the case that
52.             the viewer should be able to view more than one kind of object
53.             then superclass the two objects and view that (e.g. a list of
54.             data objects and a list of dataSelector objects might both
55.             be viewed by the same viewer, so make them subclasses of
56.             an objlist obj)
57.         constraints : dict of method : value
58.             To see if a specific viewclass object is suitable for viewing
59.             in this viewer, each of the method keys in this dict will
60.             be called and checked against the given values. The viewer
61.             will only work if the values match. For example, a viewer
62.             might only work on 2D data, so {'numDimensions':2}.
63.     displaymenu : dvmenu descriptor
64.         The top half of the "display" menu -- methods specific to this
65.         viewer (the bottom half are Display Methods from DisplayMethodBase).
66.         Menus (for DVMenu) can be defined by simple text, by a dict
67.         with any subset of the following keys (you can abbreviate them):
68.         ['text','icon','shortcut','tip','checked','WhatsThis','name']
or with lists of multiple items or nested lists to make submenus

Examples

-------

'Show error bars...' # a simple string menu item

to: 'Show error bars...', 'sh':'Ctrl+B'} # dict defined

Methods

-------

""

info =

'container': 1.0,
'verviewclass': DVCollection,
'constraints': {}, # Dict of method, values constraining what can be viewed
'vertview': 'tree' # The default vista for data in this viewer

"

displayer = {} # The portion of the display menu specific to this viewer

def __init__(self, *args, **kwargs):

    ViewerBase.__init__(self, *args, **kwargs)

    self.viewWidget = QtWidgets.QTreeWidget(self)

    self.viewWidget.setHeaderHidden(True)

    self.setWidget(self.viewObject)

    self.show() # No locatorwidgets involved here.

    def setup(self):

        ""

        Sets up the UI.

        ""

        ViewerBase.setup(self)

        if self.viewObject.name == '':
            title = 'ViewGroup Tree'
        else:
            title = self.viewObject.name

        self.setWindowTitle(title)

        self.set_tree(self.viewObject, all=False)

        self.expandAll()

        self.header().resizeSection(0, 160)

        self.viewport().installEventFilter(self)

        def set_tree(self, widget, collection, all=False):

            old_item, old_layer = None, 0 # just to begin with

            generator = collection.drill() if not all else DVCollection.drill_all()

            # set master collection item

            if not all:
                master_item = QtWidgets.QTreeWidgetItem([collection.name.name])

                master_item.object = collection

                master_item.layer = -1 # should be the case

                master_item = self.set_icon(collection, master_item)

                widget.addTopLevelItem(master_item)

            for layer, obj in generator:

                cls = obj.__class__.__name__

                if cls in ['DataSelector', 'DataSet', 'DVCollection']:

                    item = QtWidgets.QTreeWidgetItem([obj.name.name])

                    item.object = obj

                    item = self.set_icon(obj, item)

                    if layer == 0:

                        if all:

                            pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            else:

                                pass

                            el
132

widget.addTopLevelItem(item)
else:
    master_item.addTopLevelItem(item)
elif layer <= old_layer:
    parent = old_item.parent()
    parent.addChild(item)
else:
    old_item.addChild(item)
    item.layer = layer
    if item dvclass == 'DataSet':
        item = self.add_bound_selector_to_tree(obj, item)
    old_item = item
    old_layer = layer
    for x in range(2):
        widget.resizeColumnToContents(x)  
def add_bound_selector_to_tree(self, dataset, dataitem):
    layer = dataitem.layer
    for key, DSS in dataset.my_dataset.selectors.named_items():
        dssitem = QTreeWidgetItem([DSS.name, name])
        dssitem.dvclass = DSS.__class__.__name__
        dssitem.setIcon(0, self.style().standardIcon(QtWidgets.QStyle.SP_ArrowRight))
        dssitem.object = DSS
        dssitem.layer = layer + 1
    dataitem.addChild(dssitem)
    return dataitem

def set_icon(self, obj, item):
    cls = obj.__class__.__name__
    if cls == 'DVCollection':
        item.setIcon(0, self.style().standardIcon(QtWidgets.QStyle.SP_DirIcon))
    elif cls == 'DataCollection':
        item.setIcon(0, self.style().standardIcon(QtWidgets.QStyle.SP_FileIcon))
    elif cls == 'DataSelector':
        item.setIcon(0, self.style().standardIcon(QtWidgets.QStyle.SP_ArrowRight))
    return item

def eventFilter(self, source, event):
    ""
    Event filter to implement potential loading of a selected DataSelector
    ""
    if event.type() in (QtCore.QEvent.MouseButtonDblClick,):
        item = self.viewWidget.currentItem()
        label = item.text(0)
        layer = item.layer
        object = item.object
        print(layer, label, object)
        return ViewerBase.eventFilter(self, source, event)

def file_save(self, action):
    """ Saves the currently selected item into a file"
    from dataview.filehandlers.filehandlerbase import save_file
    actionName = action.objectName()
    rootlog.debug('In file_save, called with actionName {}'.format(actionName))
    item = self.viewWidget.currentItem()
    label = item.text(0)
    dvclass = item.dvclass
    if hasattr(item, 'object'):
        obj = item.object
    else:
193.     obj = None
194.     rootlog.info('In file_save; Sender is {}' . format(self.sender()))
195.     vista = self.sender().info['viewvista']
196.     if (obj is not None) and dvclass in ['DataSet', 'DataGroup', 'DVCollection', 'DataSelector']:
197.         save_file(obj, vista)
198.     else:
199.         text = 'Object is not a Collection, DataSet, or DataSelector'
200.         QtWidgets.QMessageBox.warning(None, 'Error', text)

F.7: Example LocatorWidget: LWComboBox

1. #!/usr/bin/env python3
2. # -*- coding: utf-8 -*-
3. ""
4. .. py:module:: dataview.viewers.locatorwidgets.lwcombobox
5. 
6. ========================================
7. LWComboBox Class
8. ========================================
9. 
10. LocatorWidget for a PyQt ComboBox
11. ""
12. '''
13. :Version: 1
14. :Author: Bill Dusch
15. :Date: March 24, 2017
16. '''
17. 
18. from PyQt5 import QtWidgets
19. from dataview.viewers.locatorwidgets.lwbase import LocatorWidgetBase
20. import dataview.data.locate as locate
21. 
22. 
23. class LWComboBox(LocatorWidgetBase):
24.     '''
25.     LocatorWidget for a ComboBox. This connects a Picker locator (with one dimension) to a
26.     ComboBox, changing the
Locator's index when the combobox's selection is activated. Similarly, when the index of the
Locator is changed,
the ComboBox's selection changes.

Parameters
------------
locator: subclass of dataview.locate.Locator
The locator attached to the Widget. Typically, when the index is changed, this causes the
corresponding DataSelector to update. It may also update other LocatorWidgets to change
if they correspond to the same locator. They are changed by slots on the LocatorWidget

name: str
The name of the LocatorWidget, typically the name of the dimension(s) it corresponds to.

parameters: dict
A dictionary holding parameters that correspond to options on the Widget.

widget: PyQt QObject instance
Instance of the widgetType. Parameters are stored in LocatorWidget.parameters and are set
up in set_parameters

Attributes
----------

info : dict
Information for the LocatorWidget
dimensions: int
Number of dimensions required inside the Locator

widgetType: PyQt QObject class
A PyQt QObject class or a class corresponding to the widget attached to the
LocatorWidget. This is the class,
not the instance.

Widget-Specific Attributes
-----------

None

Methods
--------

set_parameters
Sets up parameters for the widget to be used in the setup; at the least sets up
parameters attribute

setup
Sets up the widget stored inside the LocatorWidget and connects the necessary signals to
the necessary slots

postprocess
A second setup that is applied on the widget after its Viewer has been displayed

connect
Connects the widget to the slot

slot
Slot for the LocatorWidget, which updates the LocatorWidget's locator.

receive
Apply and connect the LocatorWidget to a slot on the outside.

changeIndex
Whenever the Widget's Locator's index is changed, update the widget.

# Info Dictionary

info = {
'version': 1.0, # Update version when you make substantive changes
'locator': locate.Picker  # The Locator class (as class reference) used for this
LocatorWidget
}


dimensions = 1 # Number of Dimensions of Locator

widgetType = QtWidgets.QComboBox # PyQt ComboBox is the widget

def __init__(self, *args, **kwargs):
    LocatorWidgetBase.__init__(self, *args, **kwargs)

def set_parameters(self, parameters):
    # Method to set up the parameters' keys as particular values. If blank, parameters
    # attribute set as a dictionary.
    Parameters:
    --------
    None (LWComboBox does not need parameters)
    
    LocatorWidgetBase.set_parameters(self, parameters)

def setup(self):
    # Setup method for a LocatorWidget.
    # For a LWComboBox, this adds the locator's dimension's elements to the combo box,
    # connects the slot, and connects the locator to a signal which changes the combo box whenever a
    # locator is changed
    
    self.widget = self.widgetType()
    self.widget.addItems([self.locator._dimList[0].get(x) for x in range(len(self.locator._dimList[0]))])
    
    # connect signal
    self.connect(self.slot)
    self.locator.connect(self.changeIndex)

    def postprocess(self):
        # Essentially a second setup method, to apply after the Viewer has been displayed.
        # This is useful if something needs to be drawn after the Viewer has been displayed.
        pass

    def connect(self, slot):
        # Method to connect the slot to the signal of the widget. Which signal depends on
        # the type of widget.
        # For a combobox, it is the "activated" signal
        self.widget.activated.connect(slot)

    def slot(self, index):
        # Every Locator Widget must have a slot. This slot will be connected to a signal
        # within setup
        The slot changes the index of the LocatorWidget's locator
        self.locator.index = index
        print("Picker {} index updated to {}".format(self.locator.name, index))
def changeIndex(self):
    
    """
    Whenever the Widget's Locator's index is changed, update the combo box.
    This is useful if, for example, one GUI's combobox which corresponds to the same
    locator
    is changed - so we update the other comboboxes' indices.
    """

    oldindex = self.widget.currentIndex()
    newindex = self.locator.index[0]
    if oldindex != newindex:
        self.widget.setCurrentIndex(newindex)


18. Vasudevan, R. K. *et al.* Big data in reciprocal space: Sliding fast Fourier transforms for determining


92. Fang, A. Scanning Tunneling Microscope Studies of the High Temperature Superconductor BSCCO. (Stanford University, 2009).


119. Nanonis SPM Control System. *SPECs Zurich GmbH*


125. R9plus Control Family. *RHK Technology* Available at: http://www.rhk-tech.com/r9plus/.


140. pep8. Available at: https://pypi.org/project/pep8/.


151. Classes — Python 3.6.6rc1 documentation. Available at: https://docs.python.org/3.6/tutorial/classes.html#inheritance. (Accessed: 14th June 2018)


161. 29.7. abc — Abstract Base Classes — Python 3.6.6rc1 documentation. *Python Software Foundation* Available at: https://docs.python.org/3.6/library/abc.html. (Accessed: 19th June 2018)


165. QMdiSubWindow Class. *Qt Company* (2018). Available at: http://doc.qt.io/qt-
VITA

William Dusch

EDUCATION
Bachelor of Science (Dec 2010) Physics, Stony Brook University, Stony Brook, New York.

ACADEMIC EMPLOYMENT
Research Assistant to E. W. Hudson, Department of Physics, Pennsylvania State University, May 2012 – August 2018. Research activities include: collaboration, programming, data analysis, machine learning, equipment design and procurement.
Graduate Teaching Assistant, Department of Physics, Pennsylvania State University, September 2011 – May 2017. Responsibilities include: oversaw recitations and labs, held review sessions, held extended office hours, tutored, graded homeworks, labs, and lectured in class.

PUBLICATIONS

PRESENTATIONS AT PUBLIC MEETINGS

ACADEMIC AWARDS
David H. Rank Memorial Physics Reward, Department of Physics, Pennsylvania State University, 2012
Sigma Pi Sigma, Stony Brook University, 2010.

PROFESSIONAL MEMBERSHIP