High-Speed Data Acquisition System for Nanoscale Imaging

Submitted To

Edward T. Yu
Department of Electrical and Computer Engineering
University of Texas at Austin

Prepared By

Katherine P. Anderson
Vic Frederick
Cassandra Huff
Clara Johnson
Ran Trakhtengerts
Jerry A. Yang

EE464 Senior Design Project
Department of Electrical and Computer Engineering
University of Texas at Austin

Spring 2020

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

TABLES

FIGURES

EXECUTIVE SUMMARY

The following technical report describes a novel data acquisition, processing, and visualization system for a scanning probe microscope. A scanning probe microscope (SPM) is a device that is often used to measure various aspects of a material's surface properties, such as topography, electronic, optical, thermal, electromechanical, and more, to nanoscale resolution. Traditional SPM systems are limited by low speed - up to 500 kHz - and data processing capabilities that remove the high resolution necessary to explore time-response data at such high speeds. However, many physical phenomena often can only be captured at high frequencies, so it is necessary to improve data collection and processing speeds of SPM systems. We designed, implemented, tested, and evaluated a system to augment the functionality of an existing SPM, the Dimension 3100, to collect data at up to 4 MHz speeds and process the collected data in a reasonable amount of time. The system contains four subsystems: a current preamplifier, a data acquisition system, a data processing and visualization system, and a graphic user interface. This executive summary will outline the key specifications, design solution, implementation, testing results, time and cost, safety and ethics, and recommendations for each subsystem. Due to COVID-19, we were not able to conduct system-level testing and evaluation; therefore, we have provided further recommendations on system integration for future researchers to implement.

The current preamplifier is a hardware circuit that is intended to amplify between 100 pA and 1 mA current from the SPM to voltage levels readable by the DAQ at fast enough speeds to ensure that it is not the bandwidth-limiting factor of the system. These problem statement specifications imply that the current preamplifier must have a large range of gain values, the ability to choose the specific gain value that matches the known SPM current, and a target 4 MHz speed. In addition, based on the DAQ noise specifications, we chose to set the output voltage for each gain at 500 mV to ensure that the signal was able to be discerned from DAQ or circuit noise. Together, the problem specifications and the limitations of the DAQ set the hardware requirements for the current preamplifier. In addition, an extant current preamplifier, the Ithaco Model 1211 current preamplifier, was given as a reference design to improve upon. The reference current preamplifier's specifications gave us a comparison point upon which we could test and evaluate our preamplifier.

From the hardware specifications and the Ithaco Model 1211, we designed a current preamplifier consisting of two parts: a transimpedance amplifier and a post-amplifier. The transimpedance amplifier amplifies the SPM current into a small voltage, and the post-amplifier gives additional amplification to bring the signal to 500 mV. To set the gain, we employed a novel switching circuit topology with 8-to-1 analog multiplexers, whose select line values are provided by the user on the graphic user interface and passed through the DAQ. We also used two first-order RC filters to minimize the noise produced by the circuit. To ensure safe operation, we included an on/off rocker switch and power circuitry to ensure that the board and the user would not be exposed to electrical shock when using the preamplifier. During the design process, we found that tradeoffs between the gain, bandwidth, noise, and stability limited the final design. The limitations forced us to make significant decisions on which of the four items were the most important (noise and gain), and which had to be sacrificed in the final system (usually bandwidth). To test and evaluate the current preamplifier, we simulated the frequency and noise response in Multisim 14.1. Our simulations showed that the tradeoffs and design decisions we

had made left us with severely limited bandwidth at high gains, down to 5 kHz at the highest gain setting. We also optimized several circuit parameters to reduce the noise in the circuit. While this is a far cry from the 4 MHz target, it is still significantly better than the Ithaco preamplifier's 800 Hz bandwidth at the same gain.

The current preamplifier design seemed largely circular, since our final design was an idea that we had initially come up with but abandoned early on. However, it turned out to be the best solution. Due to COVID-19, we spent additional money on prototypes that we did not have time to assemble and parts that eventually got lost in the mail. These issues led us to focus on simulation results as the main testing and evaluation tool. Future work should fabricate the current preamplifier circuit and test it in-situ with the SPM system. In addition, future work should add overvoltage protection and current suppression to ensure that spikes in the SPM current do not cause saturation in the amplifiers. In all, our current preamplifier offers conceptual proof that it is possible to tune the gain-bandwidth-noise-stability tradeoffs to obtain optimal functionality, if not the target functionality.

The data acquisition subsystem uses a NI PXIe-6124 high-speed data acquisition card (DAQ) to collect and store data from the SPM for analysis. The DAQ is the primary device that enables the entire system, as it reads multiple hardware signals produced by the SPM and stores it for data processing. Driving the entire imaging system's operation is the capability of the DAQ to sample each of the signals at 4 MHz. However, such a sampling rate demands an efficient way to manage and store a large amount of data, as average SPM tests can run from a few minutes to over an hour. The software that accompanies the DAQ is LabVIEW, which allows easy interfacing between the host computer and DAQ. The DAQ's primary source of signals is the signal access module, which contains the SPM tip bias, position, and photodiode signals. The voltage signal from the current preamplifier is also a signal that may contain useful data for specific SPM tests. However, since only four are needed at one time, the user is expected to switch the wire connections as necessary to collect the desired data. In addition, the current preamplifier relies on the DAQ to convert and pass digital input from the graphic user interface to set its gains, and the tip bias requires a 4-MHz phase-lock loop (PLL) for frequency and phase synchronization. Therefore, the data acquisition subsystem must not only acquire and store data from the four analog channels at 4 MHz, it must also output constant digital signals to the current preamplifier and a PLL.

Our design solution consists of assigning the signals to the proper ports and writing LabVIEW code to receive and store the incoming data in real-time. We provide a table for wiring and pinout in Appendix B. The analog inputs are wired to the four analog channels, and the current preamplifier outputs are wired on port 2. The phase-locked loop was not completed in time to be integrated with the primary LabVIEW code. The LabVIEW code was structured in four steps: 1. read in GUI command-line arguments, process them, and output the corresponding digital outputs to the current preamplifier, 2. set up the DAQmx task and event loop that samples the analog channels, 3. poll for the tip position signal indicating that a test has started, and 4. store every 7816 samples taken by the DAQ into a TDMS data file. Once the test is complete, the user must stop data collection from the GUI, as we found no signal that could serve as the indication of a test stopping. We chose to use DAQmx tasks as opposed to other solutions such as producer-consumer loops because they were fast enough to not overwrite the existing, unstored data. We

also chose the TDMS file type as our output file because it was human-readable, efficient at data storage, and easily interfaced with the DAQmx task.

Testing and evaluation of the DAQ system was cut short by COVID-19, which restricted our access to the research labs before we could collect any large datasets. Prior to the quarantine, we used incremental, successive steps to validate our design as we progressed. First, we connected the DAQ to the computer to verify communication. Second, we sampled noise from the analog channels and ensured that the data file was the correct size. Third, we sampled a sine wave signal from a function generator and reconstructed the sine wave with good resolution. Fourth, once we programmed the start trigger, we ran additional SPM tests to verify that data began collecting at the start condition. From the data we gathered, we were able to conclude that the DAQ interface and LabVIEW code functioned under the test conditions. Additional testing and comparing against known datasets would be the most ideal next step in testing the system.

Overall, the DAQ subsystem was completed about three weeks later than scheduled but incurred no major additional costs. However, we identified several safety considerations that were inherent in the DAQ system and were not able to be fixed. The starting and stopping sequence requires users to turn on the DAQ before the computer and turn off the DAQ after the computer to ensure that the LabVIEW drivers interfacing to the DAQ hardware would remain operational. Additional work on the DAQ interface and LabVIEW program include further testing, implementing error handling and logging, and automating the input signal switching that is currently required from the user.

One portion of the DAQ subsystem, the phase-locked loop (PLL), was intended to generate a 4 MHz signal synchronized to the phase of the tip bias. The PLL would aid in providing additional data about the phase of the signal that could then be combined with frequency to get a more robust image. While this data is not strictly necessary for our system to function, it yields an additional dimension of the data that can be explored for potential high-speed phenomena. We used an open-source existing example from NI that had many of the features we needed already, such as phase synchronization, but we also needed to implement frequency multiplication and phase tracking. We were able to implement the frequency multiplication with a small modification to the example code but realized that the 4 MHz frequency may be too high for the DAQ PLL. Due to COVID-19, we were only able to make small modifications to the example code before consolidating our project around the more significant parts of the system.

After the DAQ collects and stores data from the SPM, the data processing and visualization modules use digital signal processing to analyze and render the data for users to see.
The data processing module needed to be able to determine the position of the tip from the collected data, correlate each data point to its position, and present the data in the frequency domain. This is complicated by the sheer volume of data collected by the DAQ. Therefore, signal processing algorithms that optimize for speed and efficiency needed to be incorporated into the solution. The output of the processing and visualization modules needed to comprise viewable 3D graphics that stored and showed the frequency response–the data's evolution in time–with respect to tip position.

We divided the data processing module into five submodules: blocking, fast Fourier transform, noise thresholding, clustering, and visualization. First, to correlate the position data into discrete locations of the tip, we implemented a blocking algorithm that included a finite-impulse response (FIR) filter with windowing and padded the data so that each position contained a time series of data points that were taken at that position. Second, we took the fast Fourier transform of the time series at each position. Third, to remove as much noise as possible, we used a noise thresholding technique that replaced any data points below a certain value with a very small non-zero number. Fourth, we selected a clustering algorithm, MeanShift, to determine areas and frequencies that showed large differences between the surrounding positions and/or frequencies. For example, a position could exhibit differences in signal at different frequencies, or two adjacent positions could exhibit differences in signal at the same frequency. MeanShift focuses on areas of high activity that might be interesting to the researcher. Finally, to visualize the data, the data processing module compiles a video of the data evolving over time that is accessed by the graphic user interface so that users can examine the collected data in a human-friendly format. The data processing module was written in Python, which enabled us to utilize a variety of predefined libraries such as numpy, scipy, sklearn, and matplotlib to execute some of the computation and visualization without needing to rewrite all of the functions we needed. Due to the long processing time of the MeanShift algorithm, we also attempted to multiprocess and parallelize the code, but were not able to successfully implement it in time.

To test the data processing and visualization, we used pre-collected data from the SPM and passed it into each submodule to ensure that it produced the correct data before passing it onto the subsequent submodule. Since we did not have time to collect more than two datasets before the COVID-19 shutdown, we only used the two real datasets to evaluate the module. Our testing data showed that the module was able to produce a video from a raw input TDMS file generated an animated color map of the processed data. With respect to time and cost, the processing module was completed on time with no costs, as all of the materials we used were open-source. Because the Python libraries are open-sourced, it is possible that they may be deprecated in favor of newer versions in the future. This may cause unpredictable functionality and distorted data. Additional test data and larger datasets should be used to characterize and reduce the processing time, as it currently takes about 32 hours to process a one-minute SPM test.

The final component of the DAQ imaging system is the graphic user interface. The graphic user interface (GUI) is the primary interaction between the user and the system, and it serves as the controller to the rest of the submodules. For the system to work, the GUI should display a control panel where a user can initiate DAQ data collection, set the gain of the current preamplifier, and visualize the collected data. In addition, the GUI should provide a clean user interface that is easily navigable, intuitive, and visually appealing. These specifications guided our design and choice of programming language for the GUI.

The GUI layout is divided into five sections, where the left two sections allow the user to enter various processing and current preamplifier parameters, and the right three sections contain buttons for the user to initiate tests, run the processing module, and visualize the data. Behind the scenes, a main class calls various functions depending on the button pressed and the parameters in the text box. When the “Start Test” button is pressed, the GUI will open a command line window, run the LabVIEW executable file that contains the DAQ data acquisition and storage

code, and pass it the necessary command-line arguments to set the gain of the current preamplifier. Similarly, the “Start Processing” button initiates the Python scripts for the data processing module and passes it user-entered parameters via the command line. The GUI also has two buttons for displaying processed data. The first, “Display Blob,” displays blobs in a particular frequency bin from processed data. The second, “Generate Movie” creates and displays an animation of the data created by the data processing module. The GUI was initially built in ReactJS, a common GUI-building platform. However, a bug in the ReactJS GUI that made it unable to start processes using the command line was encountered, and a solution to the bug was never found. As a result, we switched to JavaFX, a Java library for GUI-building. For testing, the GUI simulated several common use cases and used logging to indicate whether a command was passed properly or a button’s functionality was executed. However, due to COVID-19, we were unable to rigorously test the GUI on the lab computer. While there were no additional costs associated with the GUI, there were two key setbacks in the design implementation. First, starting with ReactJS rather than JavaFX without knowledge of the command-line bug set us back about two weeks. Second, one of our team members, the expert on GUI building, became sick for two weeks. During the design implementation, to prevent data overwrite and data loss, the GUI prevents the user from initiating multiple tests at once. We also made sure to avoid plagiarizing code by using open-source Java libraries. Further work includes adding warnings and popups that contain information on each component of the imaging system.

While we were unable to fully integrate the system due to COVID-19, all of the components necessary for a fully functioning system have been developed and demonstrated to work based on the specifications, fundamental or inherent limitations notwithstanding. System integration would require interfacing each subsystem to the other subsystem as well as connecting the SPM signal inputs to the appropriate subsystems. The current preamplifier needs to be wired to the ports of the DAQ so that both the voltage output and digital select line input can communicate data. The DAQ must be wired to the current preamplifier and the SPM’s signal access module to ensure that position, frequency, voltage, and current data are all recorded for the data processing module to interpret and reconstruct. The data processing module must be provided with the GUI commands and DAQ-acquired data to produce the visual results from an SPM test. The GUI must interface with the host computer and the DAQ and processing module to pass commands and data to and from the user. Furthermore, full system testing would require successive debugging to ensure that each subsystem works as intended when fully integrated and, once fully functional, evaluating it against the original system in the lab by comparing data from the same source collected by the two systems.

Overall, we were able to design, implement, test, and evaluate all the necessary components for a fully functioning high-speed nanoscale imaging system. While we were unable to meet all of the specifications due to fundamental physical or technological limitations out of our control, we were able to construct our system by prioritizing the main goal of the project: to collect and analyze good-quality, high-speed data from a scanning probe microscope. During the implementation process, we made novel innovations in both hardware and software to achieve the project specifications, such as using state-of-the-art operational amplifiers in the current preamplifier, implementing a start trigger to indicate to the DAQ when to begin collecting data, and processing the massive amounts of collected data to identify and visualize the most

important and interesting pieces of data. We designed many safety features into our system and considered the ethics of our implementation. In addition, we completed the project on time with few additional costs: the main additional cost came in the sourcing and component purchases of the current preamplifier. Future work still needs to integrate all the individual components together, test, and evaluate the fully functioning system. Once the system is fully working, we expect it to contribute to many novel discoveries in the exploration of nanoscale phenomena.

1.0 INTRODUCTION

For our senior design project, our team aimed to design, construct, and demonstrate a high-speed data acquisition and processing interface that will interact with a scanning probe microscope and output nanoscale images on a graphic user interface for analysis. To do this, we modified an existing scanning probe microscopy (SPM) system to allow it to collect data at a rate on the order of megahertz and generate time series using that data in order to achieve high sensitivity and dynamic readings at those scales. A scanning probe microscope is a microscope that measures nanoscale phenomena. These modifications include adding a current preamplifier, an external data acquisition system (DAQ), various data processing and machine learning techniques, and a graphical user interface (GUI).

First, the preamplifier was necessary to allow the signals from the SPM to be amplified to a readable level for the DAQ. Available current amplifiers did not support the bandwidth we needed for our data collection rate of 4 MHz. As a result, we designed and implemented several prototype preamplifier boards. Unfortunately, due to COVID-19, we were unable to assemble all of our prototypes, and as a result, we were only able to test our designs through simulation. We recommend that the prototypes are assembled and tested in the lab environment in the future.

The DAQ subsystem was needed to collect and store data from the SPM for analysis. The subsystem consists of a NI PXIe-6124 high-speed data acquisition card that is controlled with LabVIEW code. While we were able to gather preliminary test data to show that the DAQ and LabVIEW code work together in basic test conditions, we were unable to test the subsystem as thoroughly as we desired due to COVID-19. When the lab reopens, further tests need to be run to prove that the DAQ works as intended.

The data processing subsystem consists of Python scripts that analyze and display data collected by the DAQ. The processing software was intended to to quickly and efficiently make sense of the data collected. While we were able to show our processing software worked on a small data set, we were unfortunately unable to vet our processing software on larger data sets due to the lab closure caused by COVID-19. Furthermore, due to the sheer amount of data our system collects, our processing software can take over a day to process a minute’s worth of collected data. To

solve this, we attempted to parallelize our processing, but the parallelized version remains largely untested. We recommend future work be done on the parallelization of the code.

Finally, our GUI is intended to provide users with an easy way to interact with our system to collect, process, and visualize data. The GUI was built in JavaFX. While the GUI was proven to work in simulated use cases, it needs to be tested with the entirety of the system. Furthermore, in the future, new warnings and safety features will have to be added to the GUI as new components are integrated into the system.

All in all, all of the components of our system have been proven to function adequately on their own. Unfortunately, the system still needs to be integrated, and the full system still needs to be vetted in the lab environment once it is possible for us to return to the lab.

2.0 DESIGN PROBLEM

We will be modifying a commercially-available SPM system--the Dimension 3100--to allow it to collect data at a rate on the order of megahertz, as well as generate time series using that data to achieve high sensitivity and dynamic readings at those scales. In general, SPM systems work by continuously repositioning a piezoelectric cantilever that interacts with the test sample surface to take nanoscale measurements (Figure 1) [1]. Normally, SPM measurement techniques are constrained by data averaging and slow sampling rates that makes close observations in time impossible. We intend to address these shortcomings by utilizing the National Instruments (NI) PXI-Based Data Acquisition (DAQ) System that can sample at megahertz speeds with top-of-the-line 16-bit precision [2]. Data acquisition systems turn real world signals into computer-readable digital signals. The SPM system has a breakout box that the data acquisition system can use to access signals generated during measurements. Some of these signals will require a large amount of amplification before they can be accurately sampled. Once samples are taken with the data acquisition, they must be filtered, visualized, and analyzed with digital signal processing algorithms. Finally, a graphic user interface will help the user interact with the system. In this section, we will discuss the design's specifications and the details of each subsystem. By the end of this project, we implemented a system that uses a DAQ to gather fast, high-quality nanoscale measurements with an SPM.

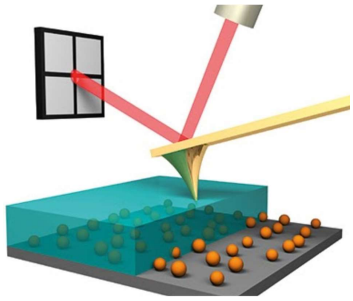

Figure 1. SPM Cantilever and Tip [1]

2.1 Current Preamplifier

The preamplifier designed for this project is intended to replace the DL Instruments/Ithaco Model 1211 Current Preamplifier with a high-speed solution that is able to meet the sampling rate of the DAQ. The Ithaco preamplifier has a bandwidth of up to 60 kHz at its smallest gain setting, which drops to 400 Hz at its largest gain setting [3]. The Ithaco preamplifier is inadequate for our design solution because the DAQ samples at 4 million samples per second, or 4 MHz. Therefore, an ideal current preamplifier should be able to amplify SPM current levels between 100 pA and 100 uA to DAQ-detectable voltages (above 30 mV) for frequencies between 0 Hz and 4 MHz with minimal noise and good stability.

2.2 Data Acquisition System (DAQ)

As a whole, the DAQ must sample and record four analog signals in phase with the current preamplifier signal and at 4 MHz frequency. The system must also output four digital signals selected by the user to be used by the current preamplifier circuit. This is accomplished with a main file and a phase-lock loop implemented separately.

2.2.1 Main System

The DAQ system must sample and record four analog signals simultaneously at 4 MHz each while outputting four digital signals to act as multiplexer select lines for the current preamplifier circuit. The DAQ needs to sample four of the five analog signals at a time between -10 V to +10 V at 4 MHz each. Because the DAQ has only four analog input channels available, analog channel zero will need to be manually rewired to the current preamplifier signal or the vertical photodiode signal depending on the kind of test being run. Rewiring the channel will not degrade performance because each kind of test requires either the current preamplifier signal or the

vertical photodiode signal but not both. Data collection must start as soon as a test starts, save the data to a TDMS file format in a location specified by the user, and stop collection reasonably soon after the end of a test. The DAQ also needs to output four digital multiplexer select lines to the current preamplifier with values determined by the user to choose the gain. By sampling four analog signals at 4 MHz each only around when a test is running and outputting four multiplexer select lines for the current preamplifier, the DAQ system will allow smooth and accurate data collection during an SPM test for later analysis.

2.2.2 Phase-Locked Loop (PLL)

The phase-locked loop (PLL) is a piece of LabVIEW code that simulates a PLL for triggering each DAQ measurement. The measurements of the signal need to be taken at consistent points in the waveform of the input signal to get usable data. If one was to measure a sine wave at inconsistent points in its period, the resulting output signal will look different in every period, even though it was originally a perfect sine wave. This causes a phase error and unusable data. The PLL is meant to output a square wave at 4 MHz with the same phase as the analog signal from the preamplifier to give the DAQ hardware an accurate and consistent matching signal and provide a sharp edge on which to take DAQ measurements for both the current from the tip and readings from the photodetectors, depending on the test. The PLL should be a part of the rest of the LabVIEW code that runs in the DAQ to synchronize measurement signals to the output from the SPM.

2.3 Data Processing

The data processing algorithm must be able to handle filtering, processing, and displaying large amounts of data in a reasonable timescale. The processing must extract and handle meaningful information from the incoming data. To do this, the algorithm needs to be able to track the position of the test as the data is constantly being sampled from the DAQ. To extract meaningful information from the massive amounts of data, the processing algorithm should also condense and simplify the output of the DAQ. Once the data is processed it is important to be able to visualize it in a well understood manner such that the user can parse through different frequencies to see the different responses. The purpose of the visualization feature is to present the processed data in a manner that allows the user to obtain the information easily and

accurately. Since the data collected changes with time and frequency this requires displaying the data in a movie format. The movie format would create a visualization method which allows the user to easily determine which frequencies displayed the most interesting features and patterns.

2.4 Graphic User Interface (GUI)

Since this system is meant to be used by a variety of researchers in a lab setting, a simple interface for the user to interact with is required. As such, the system needed a basic GUI that allows the user to quickly and easily collect data with the DAQ, process the raw data, and visualize and interact with the data. To accomplish this, the GUI must include the following features:

- A way to easily visualize and manipulate data gathered by the system.
- The ability to control the existing Dimension 3100 SPM system and manipulate its settings to facilitate data collection.
- A method of processing data using user-defined parameters.

To be considered successful, the GUI must be installed and tested on the lab computer, and its design and functionality must be approved by all team members.

3.0 DESIGN SOLUTION

Our design gathers amplified and normal signals from the SPM system at a rate of up to 4 MHz, then uses processing algorithms to find features of interest within the raw data. All of this functionality is offered to users through a friendly and easy-to-understand user interface. Our solution for the system design problem encompasses four key subsystems: the current preamplifier, the data acquisition system, the signal processing module, and the software GUI. The current preamplifier consists of two stages: a transimpedance amplifier and a voltage amplifier, and two low-pass filters to reduce noise. The data acquisition system uses LabVIEW to read in signals from the photodiodes, current preamplifier, function generator, and a position output. The signal processing module uses Python to extract desired information from the collected data and has been optimized to ensure processing time efficiency. The GUI uses JavaFX to provide a clean interface to visualize the collected data as well as control the SPM system.

3.1 Design Concept

The block diagram below (Figure 2) illustrates the interactions between each of the mentioned subsystems. The SPM outputs several signals that will be sent either to the current preamplifier or the NI DAQ. Once the data points are captured by the DAQ, it will put the collected data in a file for the signal processing module, which will then output visual and analysis data to the GUI.

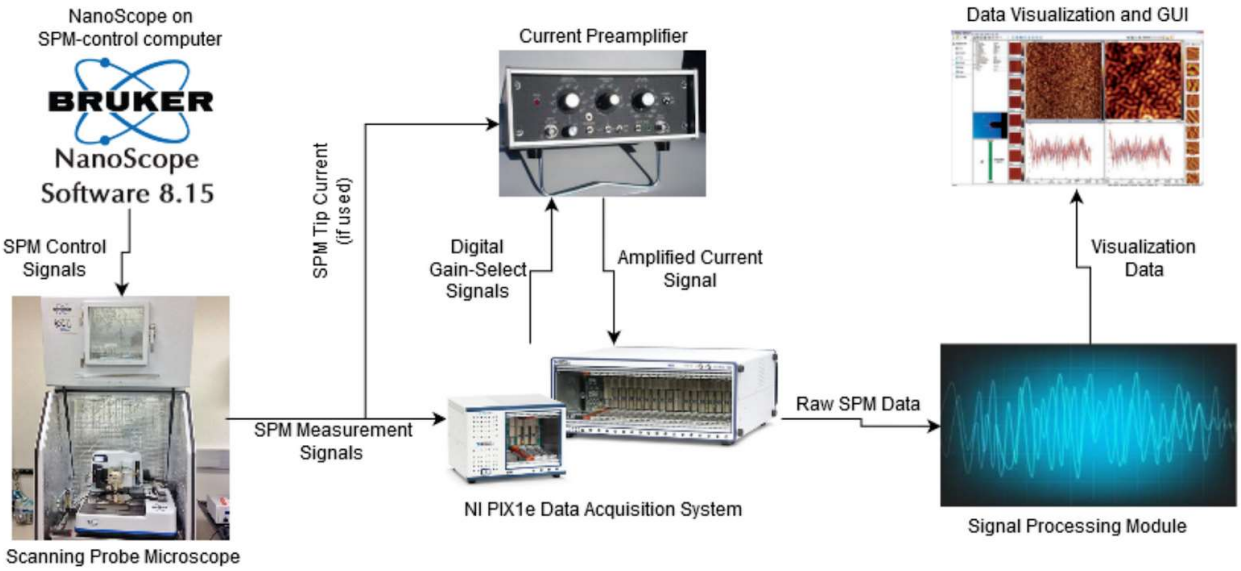

Figure 2. System Block Diagram

3.2 Current Preamplifier

Figure 3 shows a simplified schematic of the re-designed current preamplifier. The SPM current input is modeled as an AC current in parallel with a 50 pF capacitor. The 50 pF capacitor represents the BNC cable capacitance (1 ft BNC cable = about 25 pF). There are two main stages of the current preamplifier: a transimpedance stage and a voltage amplification stage. Additional RC (resistance-capacitance) filters remove noise generated by the circuit. The transimpedance stage amplifies the input current to either 10 mV or 100 mV using a variable feedback, or gain-setting, resistor controlled by switching circuitry. A feedback capacitor, whose value is dependent on the resistor, is used to prevent instability issues (e.g. ringing and peaking) associated with purely resistive circuits at high frequencies. As recommended by the manufacturer, an additional RC network is added to the positive input terminal of the transimpedance amplifier. The post-amplifier stage is a traditional non-inverting op amp circuit

that amplifies the voltage output of the transimpedance amplifier to 500 mV. The variable gain is controlled by several analog multiplexers with select lines tied to digital pins on the DAQ. Due to the tradeoffs between gain, bandwidth, stability, and noise, it was impossible with current state-of-the-art technologies to achieve the goal of 4 MHz bandwidth at all desired gains and frequencies. By carefully selecting op amps and designing filters to optimize the tradeoffs, the re-designed current preamplifier design boasts a dynamic range of 10^{-11} to 10^{-3} A and a minimum bandwidth of 5 kHz at the highest gain, an order of magnitude increase in bandwidth than the Ithaco preamplifier. In this section, we develop the theory of operation behind the current preamplifier and its various characteristics, drawing heavily on [4] and [5]. Supplemental references can be found in the annotated references list in Table A-1 of Appendix A.

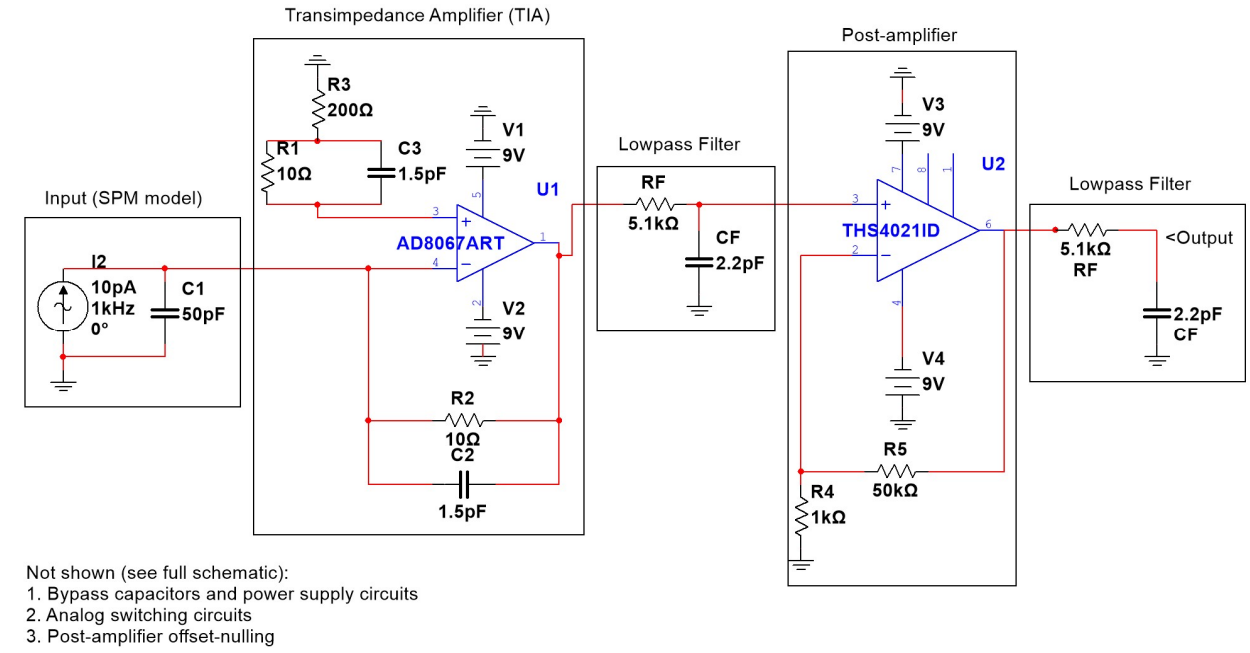

Figure 3. Simulation Schematic of Current Preamplifier

A transimpedance amplifier converts input current signals into output voltage signals (Figure 4). Transimpedance amplifiers work by application of Ohm's law: in Figure 4, the gain is given by

$$V_{out} = -R_f I_n. \quad (1)$$

Because the inverting terminal of the op amp has high impedance, a negligible current is passed into the op amp, and the remainder of the current travels through the feedback branch. When the current passes through the feedback resistance, a voltage develops across the terminals of R_f,

which sets the output voltage. The feedback resistance is called the transimpedance gain, which is different from the voltage gain. A transimpedance amplifier is limited to an inverting topology because the op amp must use negative feedback; there is no simple way to design a transimpedance amplifier that both uses positive feedback and remains stable. As a result, the transimpedance amplifier inverts and amplifies the input signal.

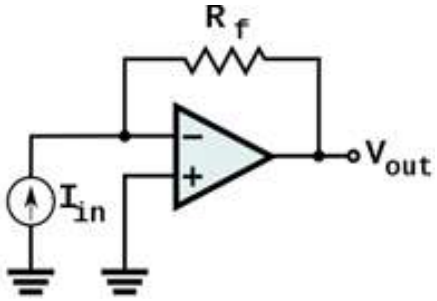

Figure 4. General Transimpedance Amplifier Circuit

Three main considerations framed the design of the preamplifier: the gain-bandwidth product, stability, and noise analysis. In linear circuits such as the preamplifier, the gain-bandwidth product is a constant that defines the maximum speed an amplifier can run at a gain of 1. Mathematically, it is defined as

$$GBW = G * f, \tag{2}$$

where *G* is the voltage gain of the amplifier and *f* is the bandwidth of the system. For example, in our circuit, we use the THS4021 as the post-amplifier device, which has a gain-bandwidth product of 3.5 GHz. Therefore, with a gain of 500, the maximum bandwidth it will be able to amplify is 3.5 GHz/500 = 7 MHz, which is above our frequency threshold of 4 MHz. In practice, the gain-bandwidth product is not a constant, as high gains and non-idealities in the circuit may lower the gain-bandwidth product. The AD8067, which is our transimpedance amplifier, has a gain-bandwidth product of 540 MHz, which means that at 4 MHz, the highest stable achievable gain is 135 V/V. However, because we are using it in a transimpedance configuration, the gain-bandwidth product does not directly apply to the circuit. Instead, for a transimpedance configuration, the maximum frequency at which the gain will be greater than 70.7% of its theoretical gain, also called the cutoff frequency, is set by the equation:

$$f_{-3dB} = \sqrt{\frac{GBW}{2\pi R_f C_{in}}}, \tag{3}$$

where R_f is the feedback resistance and C_{in} is the input capacitance. The input capacitance is the capacitance of the input cable added to the stray capacitances at the op amp terminals and is a source of noise. The gain-bandwidth relationship for transimpedance amplifiers is an inverse-square root relation, meaning that increasing the gain by a factor of four will result in halving the possible bandwidth. It is therefore exceedingly difficult to achieve full gain over the entire bandwidth for signals smaller than a few micro-amps with current state-of-the-art amplifier technology. In addition, stability and noise compensation will limit the practical frequency response to only a fraction of the theoretical response.

The transimpedance amplifier circuit, like many other amplifier circuits, also has stability issues stemming from the pole in the frequency domain that is formed between the feedback resistor and the input capacitance. This pole will cause a large spike in transimpedance gain near the cutoff frequency, which is called overshoot or peaking. Overshoot may cause ringing at the amplifier output, where a large amount of noise from harmonics and higher frequencies enter the output signal and cause it to oscillate. To cancel this pole, we add a compensation capacitor, or feedback capacitor, in the feedback loop of the transimpedance amplifier to create a zero in the frequency domain at the cutoff frequency. An estimate for the feedback capacitor can be calculated by setting the desired zero to the cutoff frequency pole:

$$f_{-3dB} = \sqrt{\frac{GBW}{2\pi R_f C_{in}}} = \frac{1}{2\pi R_f C_f} \rightarrow C_f = \sqrt{\frac{C_{in}}{2\pi R_f GBW}} \tag{4}$$

In practice, the compensation capacitor is optimized based on simulation and prototyped results, as other factors such as board wire capacitance and wire inductance add a small amount of capacitance to each trace, which may cause significant changes in the frequency response.

Noise analysis determines the contribution of various sources of noise to the circuit's performance. Noise can be treated as input-referred or output-referred, where all the noise of the system is treated as one noise source either at the input or the output of the circuit, respectively. For the current preamplifier, the dominant source of noise is the transimpedance stage. The

transimpedance amplifier has several sources of noise, including input offset voltage, input offset current, input voltage noise, and input current noise. Table A-2 in Appendix A describes the source and effect of each type of noise in detail. Input offset voltage and input offset current are DC noise sources that result from mismatches in the transistor characters at the inputs of the op amp terminals. Input voltage and current noise are frequency-dependent noise sources from both the transistors and the environment. The noise level at the output of the transimpedance amplifier is on the order of 100 uV, which places a lower threshold on the output voltage of the transimpedance amplifier. The gain of the transimpedance amplifier must then be determined by dividing the lower threshold of the output voltage (set to 1 mV) by the input current signal. Because noise analysis is often difficult and involved, we used simulation software to calculate noise characteristics for the preamplifier system. To reduce noise in the system overall, we added two lowpass filters, one on the transimpedance amplifier output and one on the post-amplifier output. The filters remove noise from higher frequencies that may be present in the signal.

After the transimpedance amplifier, the post-amplifier amplifies the output signal from the transimpedance amplifier to 500 mV by providing either 50 V/V gain for large input SPM currents or 500 V/V for small input SPM currents. Because the gain-setting resistor on the transimpedance amplifier effectively sets the frequency response of the system due to its size, the post-amplifier only needs to have a gain-bandwidth product greater than the gain of the post-amplifier times the cutoff frequency of the transimpedance amplifier. For example, a gain of 10^5 on the transimpedance amplifier (corresponding to 10 nA input reference current) has a bandwidth of only 2 MHz. As long as the post-amplifier has a gain-bandwidth product of about 1 GHz, the post-amplifier will only minimally reduce the bandwidth of the entire system. In the current preamplifier design, if the gain is set to a small enough value that the bandwidth of the transimpedance amplifier is 4 MHz, the post-amplifier will have a gain of 50 V/V so that the maximum gain-bandwidth product necessary is 200 MHz at high speeds. The THS4021 has a gain-bandwidth product of 3.5 GHz, suitable for the preamplifier.

Gain-setting is the final operating principle of the current preamplifier. The gains of the transimpedance amplifier and post-amplifier are set with analog multiplexers (also called analog switches). The MUX36S08 is an 8-to-1 multiplexer used to select between the eight different

possible gains on the transimpedance amplifier, and the TMUX6136 is a 2-to-1 multiplexer that selects between the 50 V/V and 500 V/V options on the post-amplifier. The switch on-resistance (the resistance when the switch is nominally closed) can change desired gains if the switch is placed in the feedback loop of the amplifiers, as it can add to the nominal feedback resistance already present. To prevent this, two multiplexers are used outside the feedback path to control the signal path: one between the output terminal of the op amp and the feedback node, and one between the feedback node and the next stage of the circuit (Figure 5). The op amp still drives current through its output terminal to maintain the voltage of the feedback nodes to -RfIin by Ohm's Law regardless of the analog switch on-resistance, hence the analog switch on-resistance does not affect the value of the gain anymore [6]. However, this design results in a non-planar circuit for more than two feedback resistance paths, meaning that the circuit will require at least a two-sided printed circuit board to implement.

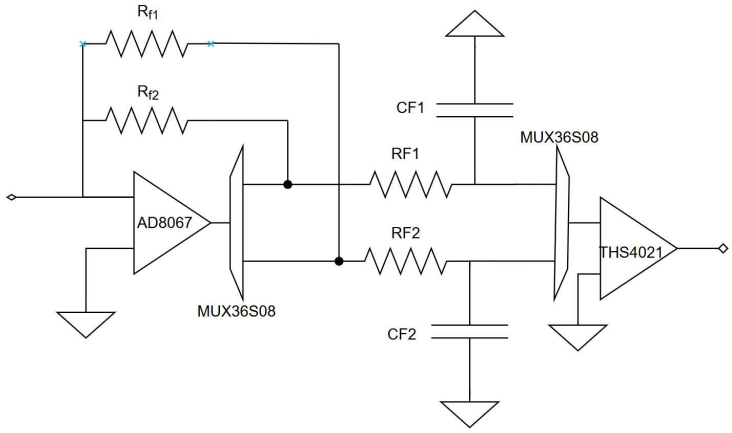

Figure 5. Gain-setting with Analog Switches

The analog multiplexers use four analog select lines to control the gains, which are controlled by digital ports on the NI DAQ. The digital ports create four constant voltages that are passed to the current preamplifier through the dual BNC jacks on the top of the PCB. The port assignments and truth table are listed in Table 1. The MUX36S08 has three select lines labeled A0, A1, and A2. The post-amplifier has one select line.

Table 1. Truth Table for Gain-select

Signal Range	Reference Current	TIA Gain (V/A)	Post-amp Gain (V/V)	A0 P2.4	A1 P2.6	A2 P2.0	Post-amp P2.7
0.1 mA – 10 mA	1 mA	10^1	50	0	0	0	0
10 uA – 1mA	100 uA	10^2	50	0	0	1	0
1uA – 100 uA	10 uA	10^3	50	0	1	0	0
100 nA – 10 uA	1 uA	10^4	50	0	1	1	0
10 nA – 1 uA	100 nA	10^5	50	1	0	0	0
5 nA – 100 nA	10 nA	10^5	500	1	0	0	1
0.5 nA – 10 nA	1 nA	10^6	500	1	0	1	1
50 pA – 1 nA	100 pA	10^7	500	1	1	0	1
10 pA – 100 pA	10 pA	10^8	500	1	1	1	1

Note. DAQ digital ports P2.4 (pin 2) and P2.6 (pin 1) should be grounded to pin 35, and ports P2.0 (pin 37) and P2.7 (pin 39) should be grounded to pin 36.

Due to the fundamental tradeoffs between gain, bandwidth, stability, and noise, we were not able to achieve the initial target goal of creating a current preamplifier that could amplify 1 nA currents to 500 mV at 4 MHz. However, we were able to improve on the Ithaco preamplifier by providing at least 5 kHz bandwidth at the highest gain setting, compared to the Ithaco's reported 800 Hz. Furthermore, we tested and optimized the passive components, the resistors and capacitors, to meet the bandwidth and stability requirements in Multisim, which we discuss later. Short from actually milling and testing a printed circuit board (which was thwarted by COVID-19), these results indicate that our re-designed current preamplifier satisfies the specifications to the greatest extent possible.

3.3 DAQ

Sampling all the necessary data and control signals will be accomplished by the NI PXIe-1071 DAQ using a NI PXIe-6124 DAQ card, shown in Figure 6. This high performance data acquisition solution will take the place of the computer-SPM interface from the existing system. With a sampling rate of 4 MHz for 16-bit conversions, we will be able to read signals moving as fast as 2 MHz and as small as 153 μV. The DAQ will be controlled using native LabVIEW software. Using the DAQ, we will output four digital multiplexer select lines to the current preamplifier circuit, and sample five analog signals using the four built-in analog to digital converters in the PXIe-6124 running at up to 4 MHz each. The five analog signals are: the signal

from the current preamplifier circuit, two photodiode signals from the SPM taken from the signal access module, a tip voltage signal taken from the signal access module, and the x-position of the SPM tip taken from the signal access module. In order to meet the physical limitation of only four analog input channels in the PXIe-6124, the analog input channel zero must be manually switched between the current preamplifier signal and the vertical photodiode signal depending on the type of test being run. The PLL is implemented in LabVIEW code and allows the sampling of the four analog signals to be synchronized to the phase of the current preamplifier signal. Using the PXIe-1071 DAQ with the PXIe-6124 card controlled by LabVIEW software, we will be able to meet all the required performance specifications.

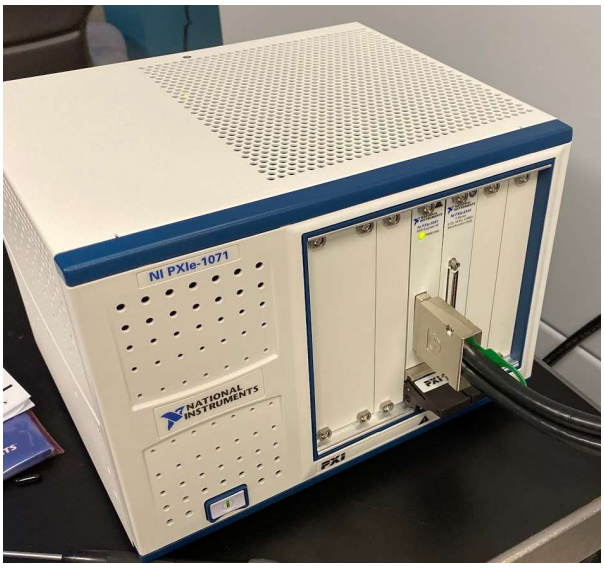

Figure 6. NI PXIe-1071 Data Acquisition System

3.3.1 Main System

The DAQ will be wired into the existing system and controlled using LabVIEW software to meet the necessary performance specifications of outputting four digital select lines to the current preamplifier circuit, sampling four analog signals at up to 4MHz each, and storing that analog data accurately to a TDMS file format that can be accessed by the data processing software. A TDMS file is organized primarily by a hierarchy of file, group, and channel. The file is whichever data file you created, a group is the set of signals taken at one time (only one group will exist in our case unless a data file was appended to), and a channel represents individual signals that are read in on available DAQ input channels. A table of the pin mapping is given in

Appendix B. Figure 7 shows at a high level what the LabVIEW software will do with the DAQ using a flow-chart format. The LabVIEW software will be called by other parts of the software system, namely the GUI, for use in the final user interface.

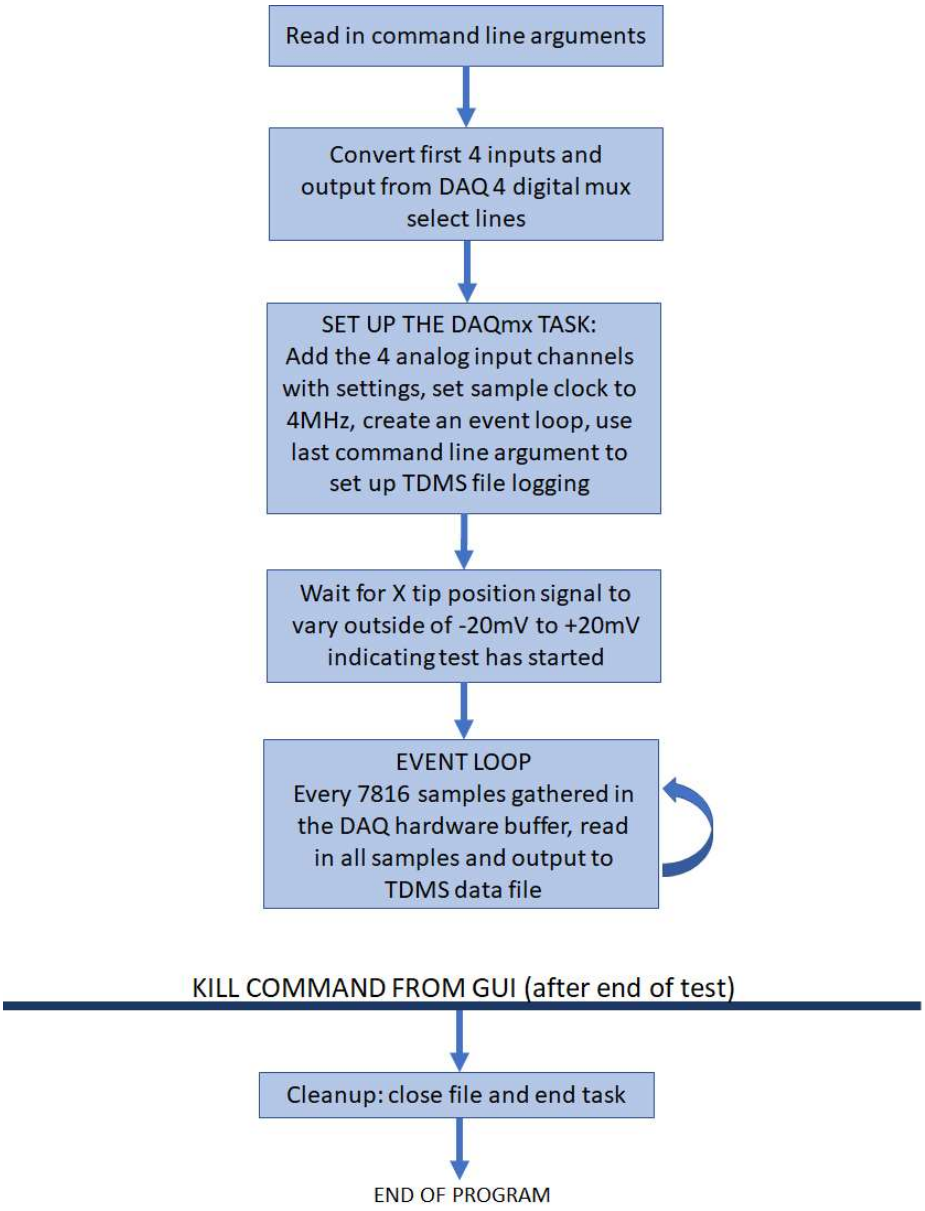

Figure 7. Flowchart of LabVIEW Data Acquisition Software

The calling software will send the LabVIEW program command line arguments indicating the gain value the user has selected for the current preamplifier circuit and the file location the user has selected for the TDMS data file. The DAQ will then output the gain value using four digital

signals to control multiplexer select lines in the current preamplifier circuit. The digital signals will be output using pins on digital port 2 of the DAQ which can be controlled to output logic 1 or logic 0 individually. These outputs will meet the first performance specification.

In parallel with outputting the digital multiplexer select lines, the LabVIEW software will also set up a software task for the DAQ by adding all four analog input channels, setting up the sample clock settings, creating an event loop, and setting up the TDMS file logging. First, each of the four analog input channels will need to be added to the DAQ task with the information that each channel is sampling a voltage signal between -10 V and + 10 V and which channel in the task corresponds to which physical channel in the PXIe-6124 DAQ card. Second, the sample clock settings are added to the DAQ task with information about which internal clock the DAQ should use to sample and how many samples per second the DAQ should take. Third, the LabVIEW software will use the fifth and last command line argument to set the file location when turning on the native TDMS file logging settings in the DAQ task. Fourth, the LabVIEW software will indicate that a start trigger will be used to start running the event loop. The final step in setting up the DAQ task is creating an event loop to trigger a block of LabVIEW code every time 7,816 samples are gathered into the DAQ hardware buffer. These setup steps will allow the DAQ task to smoothly gather data once the start trigger is recognized.

A start trigger is set up in the LabVIEW code to correspond exactly to the beginning of an SPM test. Using the data gathered in the fourth analog input channel from the SPM tip x-position, the software will start logging data when the tip begins moving as the SPM begins a test. Before a test, the tip x-position hovers in a noise window of +/- 10 mV around 0 V. During a test, however, the tip x-position signal is a consistent triangle wave from + 400 mV to - 400 mV at 1 Hz. Using this information, a start trigger can be set to trigger on the moment when the signal on the fourth analog channel leaves a window from - 20 mV to + 20 mV. Doubling the noise window means that noise on the tip x-position signal will not accidentally cause data acquisition to trigger prematurely. In this way, data collection in the event loop code will only begin once the SPM has begun its test.

Once the start trigger is received, the event loop code can start being triggered every time 7,816 samples are gathered into the DAQ hardware buffer. Each time the event loop code is triggered, it reads in all unread samples in the buffer and logs them to the previously specified TDMS file. The number is 7,816 samples because this number divides evenly into the total 2,000,896 spaces in the DAQ hardware buffer without being so small that the DAQ event loop triggers too frequently or so large that the DAQ code cannot finish reading and logging the samples before too many more samples are gathered and cause the buffer to overflow. This cycle of triggering and logging continues throughout an SPM test until the GUI sends a kill signal to the LabVIEW executable after the SPM test has finished. At that point, the LabVIEW program is terminated.

The captured signals in the end will be saved on the computer running LabVIEW as a TDMS file which can later be opened for processing by the next stage in the signal chain. Saving the data in a TDMS file meets the specification that the data processing software must be able to view and manipulate accurate data from the file format used by the DAQ. It also means that users will easily be able to open and view the raw data. The TDMS file format is very efficient at storing data in less memory due to the encoding of raw data into memory. The control of the DAQ using this LabVIEW software structure will meet all of the necessary performance specifications for this part of the system.

3.3.2 PLL

To create the PLL, we used a modified version of NI's example code for a LabVIEW PLL [7] (Figure 8). A PLL is a circuit that has a single input and a single output, with four basic parts in between. The first is a phase detector which reads two input waves and creates an error signal proportional to the difference in phase between them. The next part is a low-pass filter, which forwards a filtered version of the error signal from the phase detector to an oscillator. The design of this filter is what determines most of the characteristics of the PLL, including the stability and reactivity of the output and frequency range of the input. The oscillator, often a voltage-controlled device, is the third piece of the PLL and generates the periodic output signal with a frequency dependent on its control input from the filter. Finally, the last part of the PLL is a feedback loop from the output of the oscillator back to the second input of the phase detector. The loop can add the functionality of a frequency multiplier to the PLL, but mainly, it provides

the function of negative feedback. The negative feedback is important to maintain the stability of the output signal. As the input signal fluctuates, the filter will send a slight response to the oscillator to nudge the output signal closer to the input to stabilize the output frequency while preserving the phase difference between the two signals.

In order to maximize ease of user control, we implemented our PLL in LabVIEW. This implementation allowed any parameters to be more easily adjustable between design iterations, and later will allow parameters to be adjusted between SPM tests. The challenge in developing a PLL in software is to model all of the critical elements of a real PLL. In some ways, a software PLL is easier to implement, as the computer can give ideal and perfect data, but in another sense, we are now constrained by the limitations of software and, potentially, the machine on which the software runs. As stated before, our PLL code is based on the example software PLL from NI. We made a few modifications, which include adding a frequency multiplier and adjusting parameters to both the proportional–integral–derivative (PID) blocks and the debug interface to add user control. The output of the PLL can be realized on one of the digital output pins of the DAQ as a 4 MHz clock signal. Then, that output signal can be input back into the DAQ to be used as the clock signal for taking samples. The DAQ would be configured to trigger the measurement of all signals on the rising edge of the PLL square wave in order to eliminate phase error and measure at a consistent point of the SPM output waveforms.

Figure 8 on the next page shows the code that implements the PLL. The leftmost blocks in the code create two simulated sine waves with a random phase and a user-selected frequency, one for the input and one for the output. These signals are each fed into the section on the bottom of the loop labeled “Random Phase”. This part of the code implements both the phase detector and part of the feedback control loop, using a PID controller. The controller adjusts the phase of the output signal to keep it constant relative to the input. The top part simulates the function of the low-pass filter, but instead of simply adjusting an oscillator, like it would in a hardware PLL, there is another PID controller which implements the frequency part of the feedback loop. In the figure, the frequency controller is set to maintain a constant 4 MHz, but the infrastructure is there to allow for frequency adjustment to be implemented in the future.

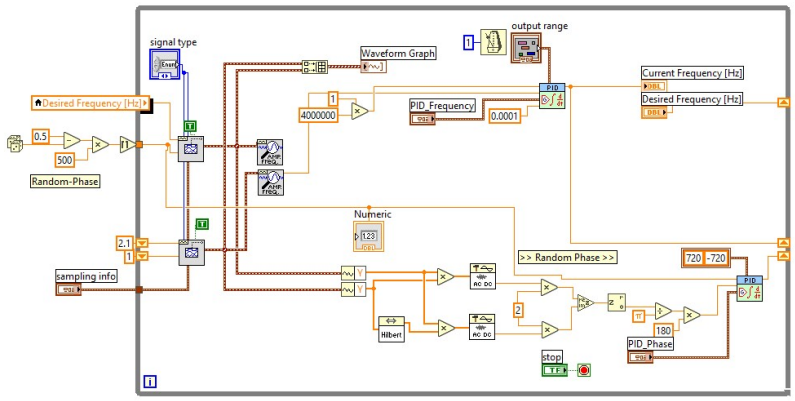

Figure 8. Modified PLL Code Block Diagram

3.4 Data Processing

The next step in our signal chain was moving from raw data to an informative representation of the test using processing. Our processing algorithm shown in Figure 9 turns the raw data from the DAQ into spatially clustered frequency responses. At the end of data acquisition, we will have access to a spreadsheet of SPM measurements and positions. The information is gathered as an adjustable X-Y spatial grid, so we will separate the data into the same dimensions, creating a three-dimensional data structure of two spatial directions and a time series at each location of space. The main hurdle to overcome in working with data from the DAQ was the constant sampling during the SPM tests with no way of knowing the exact position it was taken on the surface. We conquered this problem by using the X position sensors to dictate where the current data arises from. After the data was oriented into its correct position, a Fourier Transform was taken on all the data in a given position to create the frequency response of the location. From there, a hard threshold was applied in order to create a new data structure containing only the response likely not to be noise, and a clustering of the reduced set of points was taken to identify interesting regions of the sample's frequency response.

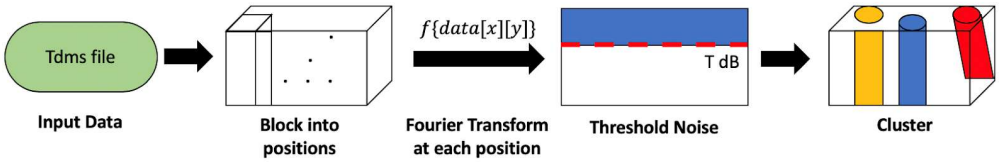

Figure 9. Data Processing Flowchart

3.4.1 Processing Submodules

Once the data was available in the processing software, the data must be spatially and temporally oriented to match how it was sampled by the SPM test. After this first step of the algorithm, we used the X position sensor accessed through the SPM's signal breakout box to establish the location of the data. The X position goes from its maximum to its minimum in a triangle wave, reflecting a trace and retrace of the SPM's tip within a user defined time on the order of 1 Hz. The X position sensor also suffers from thermal noise with a large high order harmonic. Due to the I/O limitations of the DAQ only one position sensor is available, so X-Y must both be found from the behavior of X.

First, filtered the incoming X position with a Finite Input Response (FIR) windowing filter with the following specifications. The order, which specifies the number of reactive elements contributing to the filter, was set to 200. The cutoff frequency was set to 5Hz to start attenuating the spectrum above 5Hz because this was twice as fast as the fastest trace signal we witnessed with the existing system.

- Order: 200
- Cutoff: 5Hz
- Sampling Rate: 4MHz
- Constant Group Delay

Next, we placed each X position into its appropriate pixel bin based on the number X pixels set in the GUI. The filtered and placed X position is downsampled by rate L to ensure it is monotonic during a trance or retrace. Downsampling by L refers to the use of only every Lth sample. From there, a two-coefficient high pass filter shows when the tip changes direction. We incremented the Y position everytime X goes from a negative to positive slope indicating it has moved to the next line. The user specified how many Y positions there were at the beginning of the test, so the tip moves up the sample once that last position is reached. The Y position must also decrement when the test exceeds the specified number of y positions before starting the next forward Y trace. Data with the same position was stored sequentially at that position reflecting an array evolving in time. To uniformize the data stored at this stage, zeros were added to the end of each position array, so each position array is the maximum length.

The normalized Fast Fourier Transform (FFT) of the data is taken at each XY location. The FFT is the discrete frequency version of the data. Normalizing in this case includes

- Dividing all measurements by N/2
- Using only first half of the result
- Outputting in log scale with the largest value as reference

Zero padding the data during the blocking stage made taking the Fourier transform more complicated, especially if more than half of the data was zero. The complication arises during the log normalization because the log of zero is undefined. In these cases a very small number close, but not equal to, zero was chosen as padding.

A threshold is necessary to narrow the data down the pieces that are likely to contain the most information. Due to noise in the SPM measurements, we can define a power level where data would otherwise be insignificant in our clustering algorithm. The FFT is normalized so that the largest point is set at 0 dB. As a rough number for the thermal noise, the measured SPM Signal to Noise Ratio (SNR) is adjusted to include the quantization error and noise spreading across the FFT buckets. Quantization error is the noise added to the signal by quantizing it in the ADC. Noise spreading happens in an FFT when normally distributed noise is evenly spread across all frequency bins. While the threshold does not directly translate to the relation of signal peak to noise peak, the signal is approximately uniformly distributed with the noise normally distributed. The peaks of the signal can therefore be translated to the likelihood of the noise peak. We added this correction to the threshold into the SNR as a distribution adjustment. Further testing is required to determine the usefulness of this noise model and system SNR for a given test.

At each frequency bin in the Fourier transform, the XY plane can be clustered so that the areas of activity are captured. We used the Meanshift clustering algorithm to assign a center and a radius to each point. Each point is assigned a center and a radius. The radius is originally placed at the location of the point. A radius based on the distance between each point is drawn around the center. If neighboring centers are inside this radius, then the center for that point moves to the averaged position of all points within the radius. The center is now closer to the neighboring point rather than on top of the original point. This process of moving the center continues until no new points are added to the inside of the radius. Each point goes through this process. If

multiple points have the same center by the end of the process, they are clustered. These final centers are centers of the clusters, and a researcher should gain insight in their high speed experiment by the location of the clusters for a given frequency.

3.4.2 Visualization

A video displaying the clusters at each frequency is the most beneficial to determining interesting areas. This video iterates through each frequency and displays the clusters at their location in the XY-plane while linking the amplitude of the clustered frequency response to a color on a colorbar. This allows the user to easily tell the range of values that the frequency cluster is located in while the video is played. The movie visualization can be realized using the animation and colormap functions within the matplotlib library in Python. Once the user finds frequencies they deem interesting, they can further examine a particular frequency cluster by visualizing the data as a three-dimensional object. Continuing to use the matplotlib library in Python, a 3D plot can be generated at a user-determined frequency. Once the plot is generated, the user can hover over each of the cluster centers to see the coordinates at that point.

3.5 GUI

The GUI will allow the user to easily define test parameters for the DAQ and data processing software, collect and process data, and visualize test results. The GUI is built using JavaFX, a Java-based GUI library. JetBrains' IntelliJ IDEA was used to edit code for the GUI's functionality, and Gluon's Scene Builder was used to create the GUI's layout. The code for the GUI is contained in four primary files: `Main.java, style.css, UI.fxml,` and `Controller.java`. `Main.java` is the launching point for the GUI. In it, the GUI is initialized, and the GUI window is launched. All custom styling for the GUI is contained in `style.css`. The GUI's layout is defined in `UI.fxml`. Finally, `Controller.java` contains all of the implementation for the GUI's functionality.

3.5.1 GUI Layout and Usage

The GUI, shown in Figure 10 on the next page, can be divided into two parts. The left side of the GUI is dedicated to entering parameters for collecting and processing data, and the right side allows users to collect, process, and visualize data.

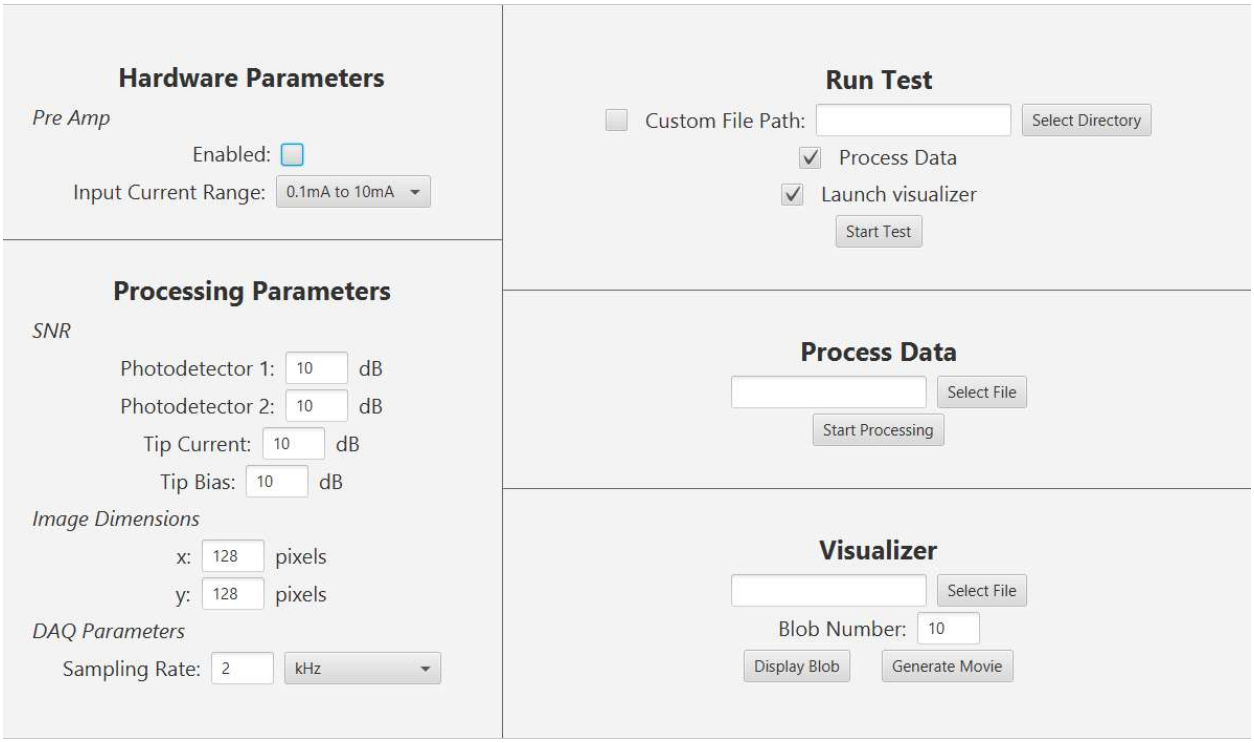

Figure 10. GUI Front Panel

The left side of the GUI lists all options and parameters that a user can use while running a test or processing data. These parameters are divided into two categories: hardware parameters used by the DAQ and the preamplifier, and processing parameters used by the processing software. In the hardware parameters section, users can choose to enable or disable the preamplifier during a test via a checkbox. The user can also set the expected current range for the preamplifier's input current. In the processing parameters section, users can set the measured SNR values for components of the SPM, the image size of the raw data, and the DAQ sampling rate for the processing software to use while it processes data.

The right side of the GUI allows users to run tests, process data, and launch the visualization scripts. To run a test, the user must first fill in any necessary parameters on the left hand side of the GUI. Next, the user must choose whether or not they want to choose a custom destination for the test results. If the user chooses to use a default destination, a folder named with the current timestamp will be created in the GUI's home folder and used. The user can also choose whether or not to automatically launch the data processing software and create a movie of the processed data after data has been collected. After this, the user can click the "Run Test" button to launch

the test. The test begins by launching the DAQ's LabVIEW executable and collecting data. When this happens, the button will become red and read "Stop Test". Figure 11 shows the GUI during this phase of the test. After the SPM test has finished, the user must click the button again to stop the DAQ. The GUI will then process the data and launch the data visualization software as needed. The raw data collected, the processed data, and any visualization files will be put into the selected destination folder. Similarly, a user can process data that has already been collected. Again, they must first fill in any necessary parameters on the left-hand side of the GUI. Then, they can select a TDMS data file to process. Finally, the user can launch the processing software with the "Start Processing" button. The processed data file will be generated and placed in the same directory as the raw data file. Lastly, the user can generate and view a movie of processed data from a CSV file with the "Generate Movie" button or view a particular cluster of data from the CSV using the "Display Blob" button.

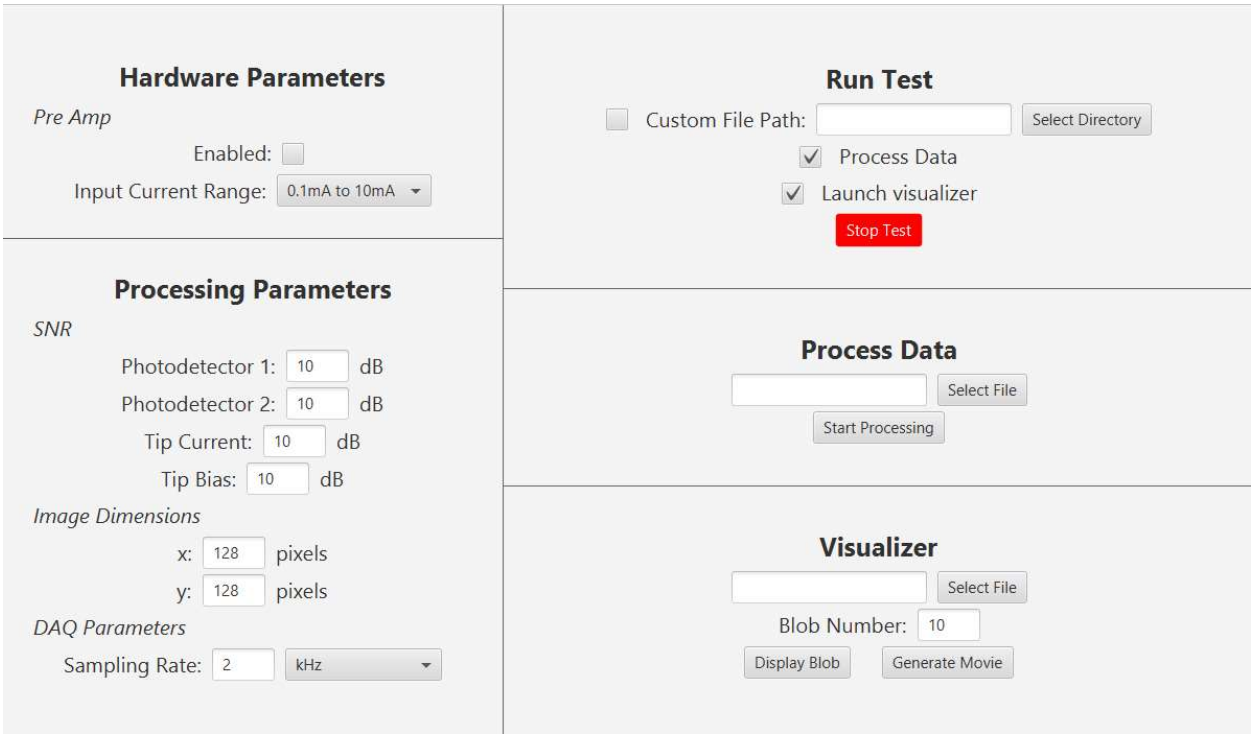

Figure 11. GUI During Data Collection

3.5.2 GUI Functionality

The functionality of the GUI is contained in the `Controller.java` file. When each button in the GUI is clicked, a corresponding function written in the `Controller` class is triggered. We discuss the specific functionality of each button. The first set of buttons handle filepaths, the "Start Test" button handles tests, and the last set of buttons handle processing and visualization.

The "Select Directory" button launches a function called `handleSelectCustomDataPath`. This function launches a file selector that allows a user to choose a destination directory. After the user selects the directory, the directory's absolute file path is written to the text box to the left of the button. The "Select File" button in the Processing Data section of the GUI launches a function called `handleSelectProcessFile`. Similar to the "Select Directory" button's function, a file selector is launched, and a user can select a TDMS file they would like to process. The file's absolute path is then written to the text box on the left of the button. The "Select File" button in the Visualizer section of the GUI launches a function called "handleSelectVisualFile," which works the same as `handleSelectProcessFile`. However, the user must instead select a CSV file, and the absolute file path of the selected file will be written to the text box to the left of this button.

The "Start Test" button launches a function called `handleRunTest`. If a test is not currently running, the GUI will change the button color and text, so the button is red and reads "Stop Test." Next, the destination path for collected data is chosen. If the user has provided a custom path, it is used. If the user did not specify a path, a default directory is generated using the current timestamp and then selected. Then, the command line arguments for the DAQ are chosen based on the parameters defined by the user. If the preamplifier is not enabled, the first four command line arguments will be `+0`, `+0`, `+0`, and `+0`. If the preamplifier is enabled, the selected input current range value will be translated according to Table A-3 of Appendix A. The final command line argument will be the directory selected earlier. Finally, the DAQ executable is launched with the command line arguments. However, if a test is currently running and data is still being collected, `handleRunTest` will reset the button to its default appearance. Then, if the user enabled processing, the processing software will be launched with the processing parameters defined by the user. These parameters will be used as is and do not require special

mapping like the preamplifier's input current range. Finally, once the processing finishes, the visualization software will be launched if enabled.

The last set of buttons control the processing and visualization modules. The "Start Processing" button launches a function called `handleRunProcessing`. This function launches the processing software and passes it the path to the TDMS file selected in the textbox above it and the parameters the user entered as command line arguments. The "Display Blob" button launches a function called `handleDisplayBlob`. This function launches the Python script `visualizing_blob.py` and passes the file and blob number selected in the Visualizer section of the GUI. Finally, the "Generate Movie" button launches a function called `handleGenerateMovie`. In this function, the file selected in the Visualizer section of the GUI is passed to `visualing_movie.py`, and the script is executed. The resulting movie is played and will be saved to the same file path as the data source file with the MP4 extension.

4.0 DESIGN IMPLEMENTATION

When implementing the system as a whole, there were many tradeoffs between subsection requirements and implementations. For the current preamplifier, a circular design process was used to optimize part choice and tradeoffs in gain, bandwidth, stability, and noise performance. With the DAQ system, the LabVIEW software had many iterations of test and debugging, as well as many changes in file-format choices and code structure meant to accommodate performance specifications. To implement the data processing software, after Python was chosen as the language, implementations based on different papers were attempted and adjusted as needed. The GUI went through a couple choices of programming languages because earlier languages were found to not include necessary functionality before the final implementation was decided.

4.1 Current Preamplifier

As mentioned in Section 3.1, the dominant issue in implementing the current preamplifier was optimizing the tradeoffs between gain, bandwidth, stability and noise. However, we went through many different designs before coming to this conclusion. Our initial design consisted of a three-stage voltage amplifier circuit in which the analog switches were placed inside the

feedback loops of the amplifier (Figure 12, next page). This design did not consist of a transimpedance amplifier; instead, it used a voltage divider to generate the voltage that would be amplified. This was problematic for several reasons. First, the bandwidth of the input voltage would be limited to the RC circuit formed by the voltage divider resistors, input BNC capacitance, and input capacitances at the first op amp. Second, each amplifier was limited to a stable closed-loop gain of 100, after which significant rolloff would occur. We went through several variations on this design until we discovered the transimpedance amplifier circuit. The transimpedance amplifier circuit solved the problem of the input-limited bandwidth and provided better frequency response. To maximize the bandwidth, we initially distributed the gain of the system relatively equally across the transimpedance and the post-amplifier; however, noise became a significant issue. The first amplifier in a cascaded amplifier system often generates the most noise on the output because its noise gets amplified through the subsequent amplifiers. As a result, it is generally better to place as much gain as possible on the first amplifier and then filter noise out before the signal is passed on. In the equal-gain system, both the transimpedance amplifier and post-amplifier contributed equally to the noise. Therefore, our final design front-loaded as much gain as possible onto the transimpedance amplifier and reserved the post-amplifier for an additional boost of 50 or 500. Thus, the transimpedance amplifier gain can range from 10 V/A to 100 MV/A. We also removed the analog switches from the feedback loop because their on-resistance would distort the precision of the gain, instead using the novel gain-switching mechanism described in Section 3.1 with two analog switches instead of one.

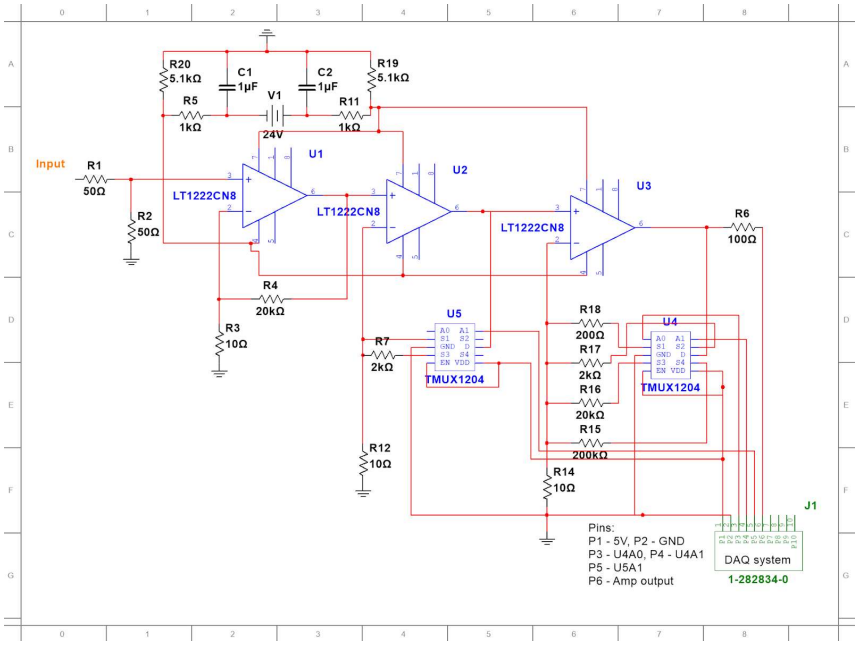

Figure 12. Early Preamplifier Design

Throughout the implementation process, we evaluated many different components on the amplifier market because it was crucial to choose the right parts for the amplifier design. The transimpedance amplifier needed to have high speed, wide power supplies to accommodate the DAQ's 10 V voltage limits, low input bias current so the signal would not be absorbed by the op amp terminals, and good noise characteristics. The post-amplifier had less strict requirements but still needed an input offset voltage that was either low enough to remain less than 10 mV after being amplified by 500 or could be nulled so that it was reduced to zero. To do this, we evaluated numerous amplifiers before settling on the state-of-the-art AD8067 and THS4021. Other amplifiers, such as the OPA846, LMH6629, LTC6268, and MAX477 had better noise characteristics but had either single-supply terminals or narrow supply rails, making them unsuitable to capture a wide range of signals. In addition, the LT1222, LT1226, LM7171, OPAx192, and THS4021 all featured wide power supply rails but had high input bias currents, making them unsuitable for sensitive measurements below 100 nA. Choosing analog switches was also important because of their wide power supply, leakage current, and number of terminals. In our case, we needed an 8x1 analog switch and a 2x1 analog switch with leakage current below 10 pA. The 10 pA limit is necessary to ensure that when a switch is "off," it does not provide current that could distort the signal. We settled on the MUX36S08 and the

TMUX6136. The MUX36S08 has a leakage current of 1 pA, and the TMUX6136 has a leakage current of 0.5 pA, making these switches the most suitable for our application.

4.2 DAQ

The DAQ system, particularly the LabVIEW code, went through several design changes before the final solution was reached. LabVIEW was chosen as the programming language because it is generally the best to program NI products, but it was difficult to learn quickly. The first designs included producer/consumer loops and logging to CSV or LVM files, but we ended in the finished design using an event loop structure and TDMS file logging because of the improved writing time. We also initially planned to include a stop trigger similar to our start trigger, but realized that was not possible using our DAQ hardware and opted to include the functionality to stop data collection in the GUI instead. For the PLL code, the design was based on example code provided by NI, then modified to suit our needs.

4.2.1 Main System

We chose the LabVIEW programming language to control the DAQ because it is the standard programming language for NI products. Programming in LabVIEW presented an unique challenge because none of our team members had used LabVIEW beyond one class we had taken four years before. We used the main tutorials provided on the NI website to begin our learning. We also found many online articles on the NI website detailing how to set up and control a DAQ task in LabVIEW. Though we ran into a few difficulties using the new language, we were able to get suggestions from various professors and eventually found an example program provided with the installation of LabVIEW that demonstrated most of the functionality we required. The structure of the LabVIEW code changed significantly once the example program was found.

In the first iterations of the LabVIEW code design we planned to use a producer/consumer structure to allow for latency in writing to the data file, but decided against it because that structure was slowing down the DAQ system such that samples were being overwritten in the hardware buffer before they could be read. Also, we initially planned to write the data to a CSV file and then a LVM file, but ended with the TDMS format because TDMS is easily human-

readable, efficient at data storage, and can be natively added to the DAQ task in the software without causing extra delays. In the first designs of the LabVIEW software, a producer/consumer structure was used with the producer loop sampling the data and pushing it into a queue, and the consumer loop reading data out of the queue and logging it to a data file. This structure was used both to prevent the hardware buffer in the DAQ from overwriting unread samples and to prevent the latency of writing to a data file from slowing down the DAQ sampling to below 4 MHz. However, we found during initial testing that the producer/consumer structure still led to both of the problems and could not be made sufficiently faster to prevent interference with accurate data collection. The previously mentioned example program installed with LabVIEW presented a solution: an event loop can be created with the DAQ task that triggers a buffer read every x number of samples while the DAQ task continues to constantly sample at the correct rate in the background. The event loop structure alleviated timing issues and allowed the DAQ code to smoothly read in data at the required rate of 4 MHz. Another change from initial LabVIEW code was the data file format. Both CSV and LVM files were tried as formats to log data, but adding code into the event loop reintroduced too much latency and samples were again lost. In the end, the example program again provided a solution. The DAQ task can be set up to natively log data into a TDMS file as it is read from the hardware buffer. Once a Python library was found that allowed our data processing software to read data out of a TDMS file, the format was finalized. Switching to TDMS files added two bonuses: TDMS files are very user friendly so raw SPM data is easily viewable, and TDMS files store data very efficiently so our large data files are a bit smaller than they would have been otherwise. Changing the structure of the LabVIEW code to an event loop structure and changing the data file format to TDMS means that the end user experience is accurate and smooth.

At the beginning of the DAQ work, we planned to use some signal from the SPM system as a trigger to stop data gathering at the end of a test. After consultation with NI engineers, the implementation of this proved impossible with our hardware, so we decided instead to have the GUI ask the user to stop the test once finished. We had initially planned to use a signal to stop data collection in the same way as we start it. An idea was formed that met all constraints to use when the SPM tip X position stopped varying by more than +/-10 mV as our stop trigger. However, detecting this moment is very hard to implement with the DAQ hardware in our PXIe-

6124 DAQ card. After consulting with two NI engineers, we realized that this kind of stop trigger is, in fact, impossible to implement with our hardware. We then needed a solution besides a stop trigger because no other SPM signals that had clear behavior associated only with the end of a test could be read into the DAQ. We decided that having the user press a button in the GUI after a test finishes to end the LabVIEW program was the best solution. Though a small amount of extra data will be collected at the end of each test, the GUI button is better than our other option of simply having an automated timeout because SPM tests vary in length. The GUI button will still allow users to end data collection in a reasonable amount of time after an SPM test finishes.

In the final design of the DAQ system, LabVIEW software is used with an event loop structure to log data into a TDMS file during an SPM test. To stop data collection, there is a button in our GUI that will allow a user to end the LabVIEW program. The end product of the DAQ system meets all the design specifications and allows the user to accurately gather the correct data during an SPM test while logging that data into a file that can later be used by our data processing software.

4.2.2 PLL

Figure 13 on the next page shows the PLL LabVIEW file with our modifications highlighted in the yellow box. The modification sets the target frequency of the frequency PID controller to 4 MHz. Because the PID controller in a traditional PLL in LabVIEW takes in two inputs and effectively tries to match the output frequency to the input frequency, all we had to do was modify one of the terminals to input a constant of 4,000,000 to set it as our desired frequency. However, to ensure that the PID would be able to adjust to various frequencies of the input signal, we added a multiplier on the 4,000,000 constant in case future users wanted to modify the output frequency of the PLL signal.

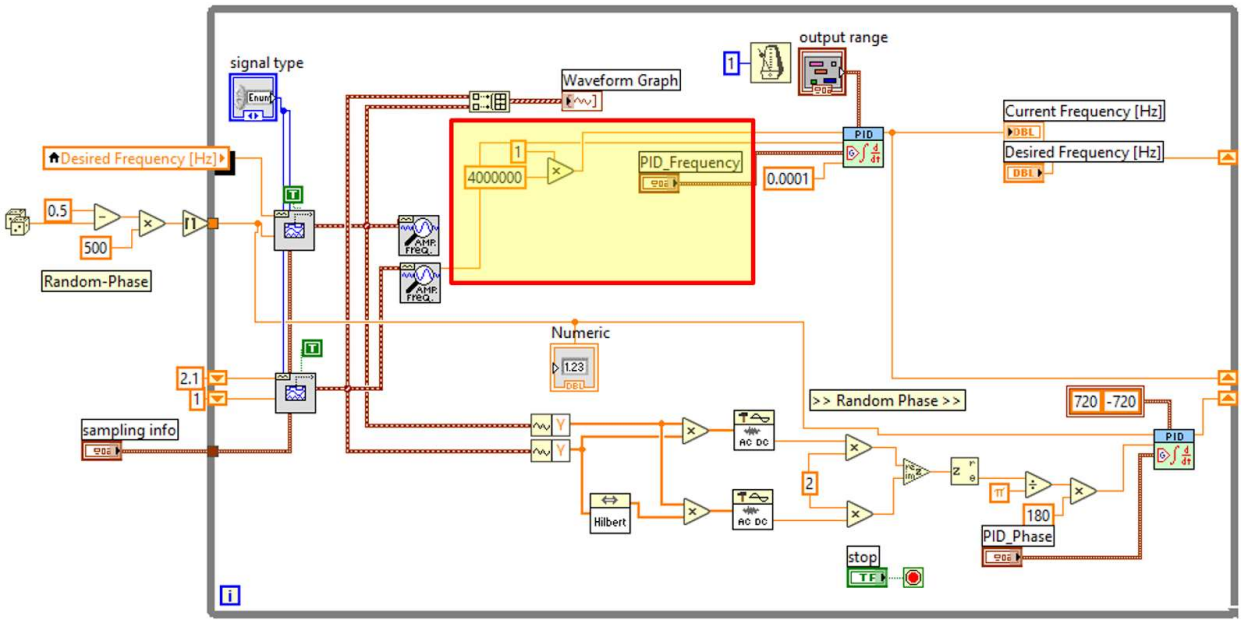

Figure 13. PLL LabVIEW Code with Highlighted Modifications

The code in the figure is separate from the rest of the LabVIEW code because it is easier to develop and debug with more display outputs and manual controls. This form allows for simpler manipulation of individual parameters, instant feedback to the programmer, and less load on the computer when running. In the future, this code could be slimmed down to include only the direct PLL functionality and cut out the real-time outputs to the user. All of the manual controls mentioned previously will instead be controlled by the DAQ code, calculated automatically, or directly governed by the user through the GUI.

4.3 Data Processing

The data processing underwent several iterations ranging from changes in language to changes in the structure of the code. We decided to use Python instead of MATLAB to write the data processing code due to the added speedup from the Python SKlearn library. To obtain information regarding the Y position, the XY positioning code had to manipulate the X position waveform. The thresholding algorithm was adjusted after performing the risk reduction to implement an initial thresholding algorithm which had sparse information regarding the method. After running the sequential processing code we found that a significant speedup would occur if portions of the data processing were multi-processed, despite the added complexity. Finally, the

visualization was adjusted from having just the points on the plot to being capable of visualizing the specific values at each cluster to better obtain information regarding the data points.

The algorithms to block, Fourier transform, threshold, cluster, and visualize data stored in TDMS files by the NI DAQ were implemented in Python3. Python was our final platform after staying open during the first half of our development because of the available libraries for clustering and visualization over its competitor MATLAB. When deciding between Python and Matlab, we compared the MeanShift algorithm because our implementation requires a sort that is $O(n^2)$. After testing both languages, Matlab was found to have a $O(n^2)$ algorithm; whereas, the SKlearn library in Python had a O(nlogn) time. Because the program runs this algorithm 18000 times in our small one minute example, and encompasses an X pixel by Y pixel number of points per run, we chose to implement the remainder of the processing in the same language. Useful tests may extend for much longer than one minute, so big-oh is a crucial consideration for the processing.

Once the language had been chosen, positioning the data was the next implementation hurdle. As described in Section 3, the data comes into the processing as a continuous channel that must be broken up into positions. The data placed at the same position will be treated sequentially as a time series. These discrete X-Y position determinations must be made off of the analog position waveform for only the X position. The tip moves in a raster pattern sliding across a row and moving back across that same row again before transitioning to the next column and repeating the process. At the last row, the tip moves back up the scanning pattern to finish the scan. The consistent behavior allows us to detect when a new Y position is reached by registering the X position as it moves left to right as indicated by the sign of the X position waveform. Unfortunately, the X position waveform is not monotonic when it is moving in either direction. The X position waveform experiences a small high-order nonlinearity and thermal noise. To isolate the sign change further, the signal is downsampled. The downsampling lowers the chances of a false new line, and lowers the amount of times we have to check for a new Y position because the data will not be sensitive to delays in updating the Y position under our chosen clustering analysis. Positioning is now one of the most reliable aspects of our processing chain, and it is suited for future works if different analysis methods are chosen after our project.

The thresholding aspect of the code was originally to be based off of an example code from [8] as one of the risk reduction activities. After attempting to replicate the code from [8], not enough information had been provided to use their method. Thus, we used a simpler but effective method of thresholding the data, which finds the threshold using an assumption that the noise produced by the amplifier was negligible compared to the noise produced by the SPM and the DAQ. When the thresholding algorithm was being integrated into the subsequent processing algorithm, we found that it would be best to return the data structure differently from the original 3D array being passed. Therefore, we decided to reformat the output data as a 3D array which iterates through frequency bins instead of X,Y points. We integrated this into the thresholding loop to minimize time delays going into the next segment of the processing algorithm.

Adding parallelism to the code was necessary to process large datasets on the order of megabytes or terabytes in a reasonable amount of time. After some initial testing of the code on real datasets, we decided to add multi-processing to the already-working, but sequential, data processing code in order to reduce the runtime. We decided that using a "pool" of processes that tasks could be submitted to was better than explicitly starting multiple processes, since the multi-processing was added to a finished product. Many of the long-running for-loops in the code were turned into multi-processed code using this pool of processes. This allowed multiple indices of the data to be manipulated in parallel, reducing the amount of time it took to run each for-loop by a factor equal to the number of CPUs on a particular machine. It was difficult to find ways to pass data back and forth between processes without using too much extra memory. However, by using a new pool of processes for each multi-processed for-loop, Python was able to free up unused memory faster. In the end, the added complexity of multiple processes is much worth the gained speed-up of the processing code.

A colormap was used to easily access the information about frequencies in the movie format. The visual color will give the user a quicker way of determining which values are interesting without having to individually read each value. The single images produced allow the user to see the x, y, and z components of each cluster at specific frequencies. We later determined it would be convenient to be able to hover over each cluster and see the x, y, and amplitude of the

frequency response, because the specific values on the plot were not easy to read. Hovering over each cluster will let the user easily obtain information regarding the data values.

4.4 GUI

During the design process, we had to decide what language to write the GUI in, how to structure the GUI's implementation, and what the GUI should look like. We initially meant to use JavaFX. However, due to requirement changes, we decided to instead use ReactJS and Electron to build the application, and we built our first prototype using these tools. Unfortunately, this prototype was abandoned, and our final GUI was built in JavaFX as originally planned. The implementation of the final JavaFX GUI was kept simple to facilitate future maintenance and expansion. Finally, our GUI layout evolved through several iterations to allow its design to be simpler while allowing users greater flexibility in their defined parameters.

The first step in the GUI's implementation process was choosing a language in which to write the GUI. The team initially decided to use JavaFX, as we planned to do all of the data visualization in the GUI. Because of the amount of data we would need to display, we decided that we would need to use a compiled language, like Java, for the GUI to ensure it would be fast enough to handle all the data. As several members of the team had some experience using JavaFX, we decided to use it. However, the GUI and data visualization were later decoupled, as we decided that the visualization component was better suited to our data processing team. Therefore, the requirement for our GUI to be written in a compiled language was dropped. Since the GUI team had extensive experience building JavaScript GUIs, it was decided that the GUI would instead be built using a combination of ReactJS--a JavaScript framework--and Electron--a framework that allows a JavaScript application to run as a desktop application. A mockup of the GUI layout was created, and a prototype of the GUI was implemented in ReactJS. Unfortunately, a bug that prevented the ReactJS GUI from launching child processes was encountered. Although our research indicated that an application built with ReactJS and Electron should be able to launch a child process, our particular configuration was unable to. Unfortunately, a solution to this issue could not be found. As a result, the ReactJS GUI was scrapped, and the GUI was built using JavaFX.

Since the team did not have much experience developing in JavaFX--and since it is unlikely that future users and developers of this system will have extensive JavaFX development experience--the structure of the GUI was kept simplistic. First, IntelliJ IDEA was used to generate a "Hello World" JavaFX application. This application was used as a basic template for the structure of our GUI that we then modified as needed. Due to this approach, the GUI consists of four main software files: a controller class written in Java, a main program written in Java, a style sheet written in CSS, and a layout file written in FXML. First, the controller class contains all code related to the functionality of the GUI. This includes handling all button clicks, file selections, and parameter parsing. Next, the main program handles the initialization of the GUI and launches the GUI's window. The CSS class holds basic styling needed to make the GUI easy-to-read and attractive. Finally, the FXML file contains code for the actual layout of the GUI and serves to decouple the layout of the GUI from the functionality implemented in the controller. IntelliJ IDEA was used to write the main file, the controller class, and our style sheet. However, to simplify designing the GUI's layout, Scene Builder was used to create our FXML file.

Lastly, the GUI layout went through multiple iterations before we finalized it. Figure 14 shows our original GUI mockup. The mockup organized all the test parameters that the user needed to be able to enter and provided a way to collect, process, and visualize data without having to manually launch the corresponding scripts via the command line. However, this design was complex and included extra features, such as a way to minimize and maximize parameter sections and live progress trackers.

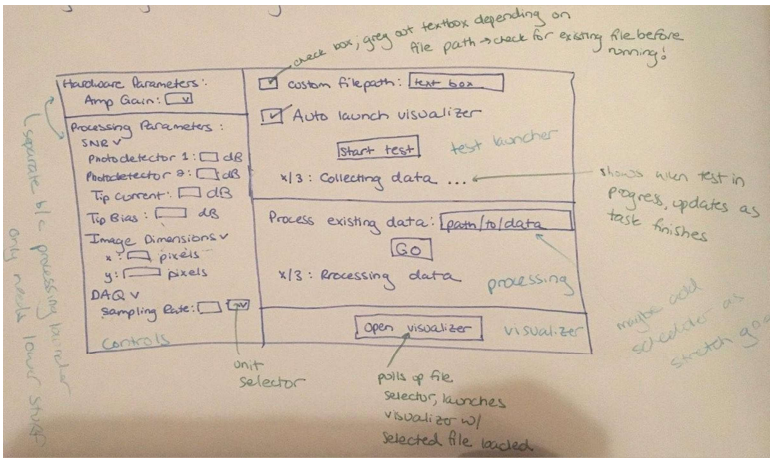

Figure 14. Original GUI Mockup

Using this mockup, the first prototype of the GUI, shown in Figure 15, was created using ReactJS and Electron. While developing this prototype, the extra features discussed were dropped, as they added unnecessary complexity to the GUI's implementation that would make maintenance of the GUI harder than necessary. However, as previously discussed, this prototype was ultimately abandoned, and JavaFX was used for our final product.

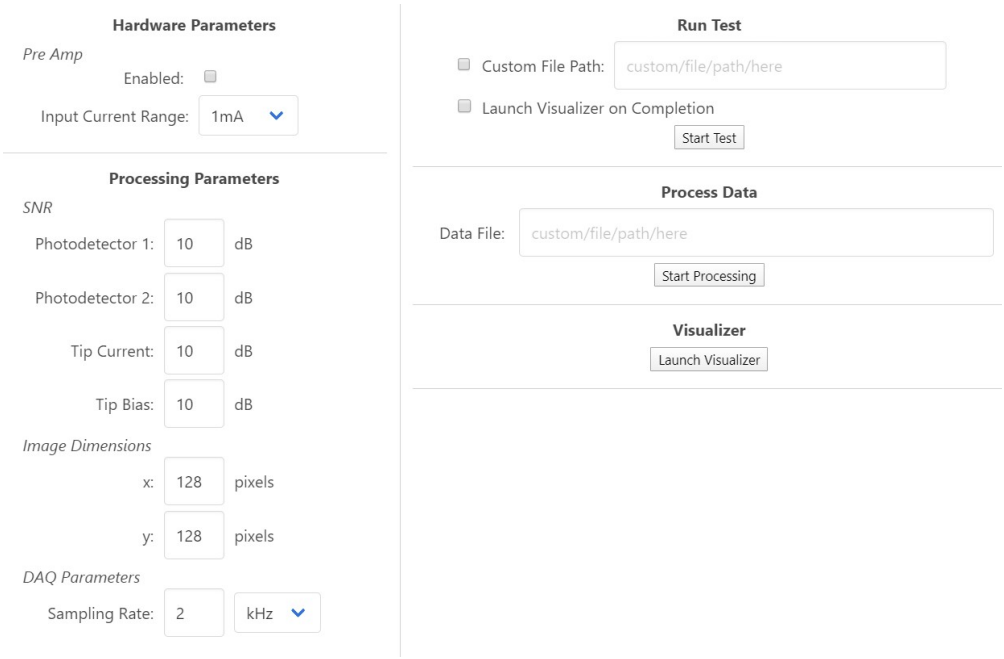

Figure 15. ReactJS GUI Prototype

Finally, the JavaFX version of the GUI was created. First, a default "hello world" JavaFX application was generated by IntelliJ IDEA. Next, the default FXML file was modified with Scene Builder to create the layout for the GUI. The layout was largely copied from the ReactJS GUI. However, a few new options were added, and the format of a few fields was changed for ease of use. Finally, the default controller class was modified to add functionality to the buttons on the GUI. Figure 10 in Section 3.5 shows the resulting final product.

5.0 TEST AND EVALUATION

An important aspect of the design project is having a test plan that will ensure that the subsystems work accurately and effectively as a cohesive system. To test the current preamplifier, simulations were run in Multisim. The DAQ system was tested first using a

function generator, and then using signals from the real SPM system. The data processing software was tested by coding individual testbenches for each function before putting the entire program together to test. The GUI was tested manually using simulated user use cases. Due to the COVID-19 outbreak, testing of the whole integrated system is delayed.

5.1 Current Preamplifier

Due to COVID-19, instead of fabricating and testing an actual printed circuit board, we used Multisim 14.1 to validate and optimize our current preamplifier in Figure 3. For frequency and noise response, we conducted simulations using Multisim's AC Sweep and Noise SPICE tools to generate amplitude and phase curves, as well as noise values for the current preamplifier. The feedback capacitor and lowpass filter values were reoptimized for each gain setting. The simulation results show that our design solution improves on the Ithaco preamplifier by providing larger bandwidth at comparable noise. However, the limitations of the model may cause an eventual decrease in the current preamplifier's performance to about the same as the Ithaco in a functional board.

Table 2 and Figure 16 on the next page show the frequency response of the current preamplifier. An AC sweep was run from 1 kHz to 5 MHz for each gain. C2 and C3 are optimized to minimize the overshoot and bandwidth in the transimpedance amplifier. An input source capacitance of 50 pF (~2 ft BNC cable) was included in the model.

Table 2. Frequency Response of Current Preamplifier

Signal Range	Reference Current	TIA Gain	C2	C3	TIA Bandwidth	Post-amp Bandwidth	Overshoot
0.1 mA – 10 mA	1 mA	10^1	1.5p	1.5p	>4 Mhz	>4 Mhz	<0.5%
10 uA – 1mA	100 uA	10^2	1.5p	1.5p	>4 Mhz	>4 Mhz	<0.6%
1uA – 100 uA	10 uA	10^3	1.5p	1.5p	>4 Mhz	>4 Mhz	4%
100 nA – 10 uA	1 uA	10^4	2.2p	2.2n	>4 Mhz	>4 Mhz	9%
10 nA – 1 uA	100 nA	10^5	.75p	68p	3.1 MHz	2.7 MHz	4%
5 nA – 100 nA	10 nA	10^5	.75p	68p	3.1 MHz	1 MHz	4%
0.5 nA – 10 nA	1 nA	10^6	0.3p	75p	750 kHz	200 kHz	None
50 pA – 1 nA	100 pA	10^7	0.3p	0.3p	55 kHz	35 kHz	None
10 pA – 100 pA	10 pA	10^8	0.3p	0.3p	5 kHz	5 kHz	None

The amplitude response curves, generated from Multisim, were plotted and used to calculate the simulated cutoff frequency for each nominal gain. The cutoff frequencies were reported in the table. The current preamplifier shows large bandwidths, with some exceeding 4 MHz at low gains, as evidenced by the amplitude response curve in Figure 16. However, for nominal gains greater than 5 MV/A, the current preamplifier began to suffer significant bandwidth loss from 4 MHz to 5 kHz. For comparison, the Ithaco preamplifier's bandwidth loss was from 60 kHz to 400 Hz. The current preamplifier also shows good stability in the frequencies of operation, before the cutoff frequency. In a stable system, the phase should drop to zero after the cutoff frequency, which was true for all gains.

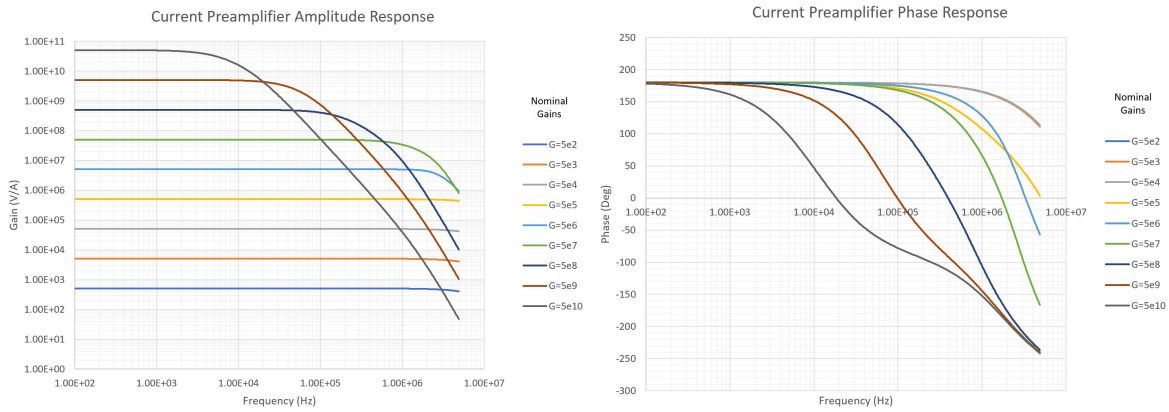

Figure 16. Current Preamplifier Simulated Frequency Response

Table 3 and Figure 17 on the next page show the noise characteristics of the current preamplifier as reported by Multisim. The noise appears to peak at the transition when the transimpedance amplifier is set to a gain of 100 kV/A, which may indicate that when the transimpedance amplifier gain resistor is close to within an order of magnitude of the post-amplifier back resistor, the noise may be significantly amplified. However, the overall noise characteristics remain close under 50 mV at low gains and under 150 mV at high gains. While this almost certainly can be minimized further, it is still below the desired signal of 500 mV on the output.

Table 3. Noise Response of Current Preamplifier

Signal Range	Reference Current	Total Gain (V/A)	RF	CF	Bandwidth	Integrated Total Noise
0.1 mA – 10 mA	1 mA	$5x10^2$	5.1 kΩ	2.2 pF	>4 Mhz	2 mV
10 uA – 1mA	100 uA	$5x10^3$	5.1 kΩ	2.2 pF	>4 Mhz	2 mV
1uA – 100 uA	10 uA	$5x10^4$	5.1 kΩ	2.2 pF	>4 Mhz	3 mV
100 nA – 10 uA	1 uA	$5x10^5$	5.1 kΩ	2.2 pF	>4 Mhz	20 mV
10 nA – 1 uA	100 nA	$5x10^6$	5.1 kΩ	4.7 pF	2.7 MHz	37 mV
5 nA – 100 nA	10 nA	$5x10^7$	5.1 kΩ	15 pF	1 MHz	115 mV
0.5 nA – 10 nA	1 nA	$5x10^8$	5.1 kΩ	0.1 nF	200 kHz	120 mV
50 pA – 1 nA	100 pA	$5x10^9$	5.1 kΩ	0.47 nF	35 kHz	87 mV
10 pA – 100 pA	10 pA	$5x10^{10}$	5.1 kΩ	2.2 nF	5 kHz	68 mV

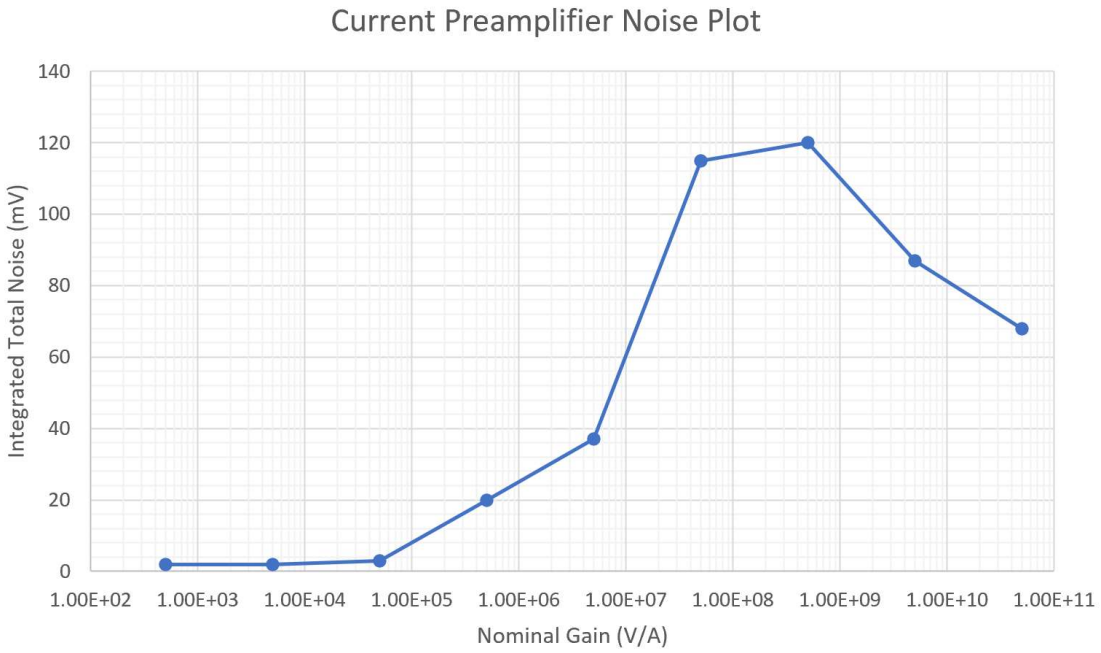

Figure 17. Current Preamplifier Simulated Noise Plot

There are several limitations with the Multisim model. The inability for Multisim to model the behavior of the analog switches we selected makes it difficult to understand their effects on noise. Because the analog switches are directly in the signal path, they can contribute significant amounts of noise and are the most important noise source that is not present in the model. In addition, the Multisim model does not account for offset-nulling circuitry or bypass circuitry. Offset-nulling circuitry on the THS4021 would enable the reduction of input offset voltage, increasing the signal quality at low frequencies. Bypass circuitry is circuitry that ensures that

constant power is continuously supplied to the integrated circuits. It is relatively inconsequential to the current preamplifier response but is closely related to the non-ideality of the power sources. The model assumes that all sources are ideal and that all resistors have exact values, which is not the case in real life. All components have various tolerances that must be accounted for. In addition, the printed circuit board may have parasitic capacitances that may cause significant differences between the simulated bandwidth and actual bandwidth if not designed properly.

5.2 DAQ

The DAQ was tested first by connecting the system to the computer and verifying communication, then by sampling noise on all four analog input channels and verifying data file size. Then we sampled a sine wave at three different frequencies on each analog input channel and verified that the wave was read in correctly, and then we tested the start trigger by running the DAQ system with the SPM system and verifying when data collection began. Ideally, one more kind of test can be performed, once COVID-19 has passed, with the SPM running tests on samples with known or expected responses so that the DAQ data collection can be comprehensively verified. The PLL was tested only in simulation due to COVID-19.

5.2.1 Main System

The initial DAQ system testing began when the DAQ was connected with the computer and we ensured that communication occurred correctly. The very first tests of the controlling LabVIEW software were ensuring that the file sizes were what we expected after running the software for a particular amount of time and using a multimeter to check that all digital multiplexer select lines were correctly 1 or 0 based on user input. During the initial connection between the DAQ hardware and the lab computer, there were some communication issues, but after an extra NI driver was installed, communication between the two devices was consistent. The first tests of the analog input sampling involved sampling noise on all four analog input channels for a timed interval (usually 10 seconds) and then checking to see that the output data file was the expected size for data collection of that length of time. This test initially showed the error that the DAQ was not taking enough samples because the data files were significantly smaller than they should have been. However, after some debugging and design changes, this test yielded correctly sized

data files. In parallel with this file size test, the digitally output multiplexer select lines were also tested by running through all sixteen possible 1/0 combinations on each select line and verifying correct voltage output from the DAQ with a multimeter. With comparatively minimal debugging, the LabVIEW software was passing this multimeter test. These initial tests helped uncover early bugs in the controlling DAQ software so that later stages of programming were smoother.

Once the system passed initial tests, more stringent tests were devised to ensure that each analog input channel correctly read in a sine wave generated by a function generator at different frequencies. Then, we began testing while the SPM system was running to verify that the start trigger worked correctly. To verify that data collection on each of the four analog input channels occurred often enough and accurately, we began to sample sine waves of three different frequencies from a function generator on one channel at a time and then plotting the raw data, as in the example shown in Figure 18 on the next page, to verify that no distortions or data loss occurred. With minimal extra debugging, the DAQ system passed these sine wave tests almost immediately. Though we planned to later also run tests using very high-frequency sine waves to strenuously test 4 MHz data collection, we were unable due to lab access closing because of COVID-19. Once accurate data collection was verified as much as possible without the SPM running, we began running the DAQ while the SPM was running a test. Before access to the SPM system was cut off, we were able to verify that the start trigger correctly begins data collection at the point an SPM test begins. We did this by starting the LabVIEW software, and then waiting an arbitrary amount of time before starting an SPM test and watching the LabVIEW software begin data collection immediately after. Though more thorough testing will need to be done using the SPM system and the DAQ system in conjunction, these finished tests are enough to be fairly confident that the DAQ system will perform within design specifications.

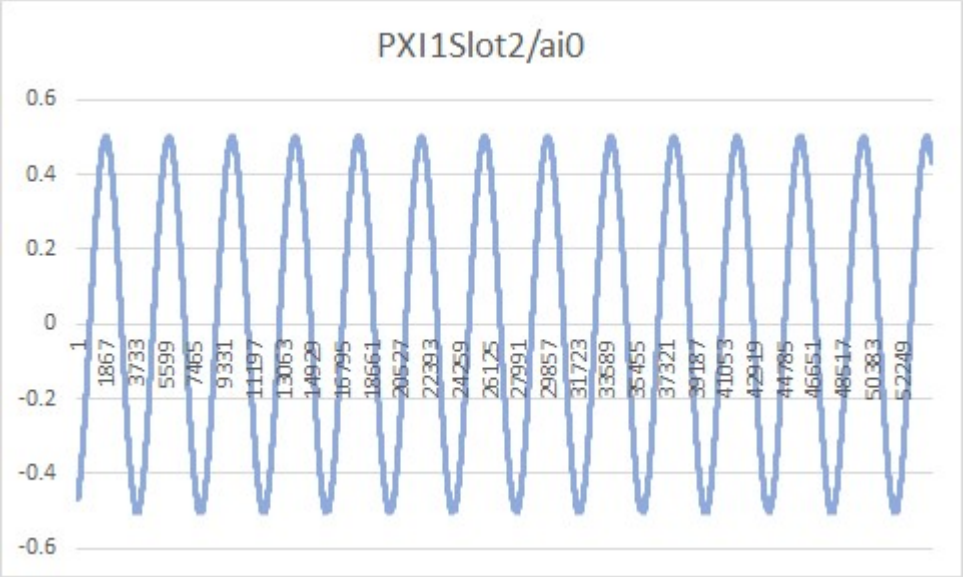

Figure 18. Sine Wave Test on DAQ Analog Input Channel 0

In the future, once the lab at Pickle Research Campus reopens, formal testing against known datasets will need to be run with the DAQ system and the SPM system running. These tests would involve taking data on a sample with a known response using the SPM and then verifying that the raw data taken during the test by the DAQ system is as expected. Preferably, this can be done on several samples. After this last testing step, the functionality of the DAQ system would be fully verified and guaranteed (within tolerances) to be correct.

Tests of the connection between the DAQ and the computer, the data sampling rate on noise, the data sampling rate and the data sampling accuracy on sine waves, and the start trigger feature have verified the DAQ system to a level of reasonable expectation of correct functionality. Extra tests on known samples using the SPM system will further ensure correct data collection.

5.2.2 PLL

In the testing phase, the PLL was not successful in outputting a consistent 4 MHz square wave. The test involved feeding a computer-generated square wave into the input, with the output settings configured to produce a 4 MHz square wave with the phase held constant to maintain a constant phase difference. The output of this test was not a square wave, and can be seen in

Figure 19 below. The red waveform is the sample input waveform, and the white waveform is the PLL output.

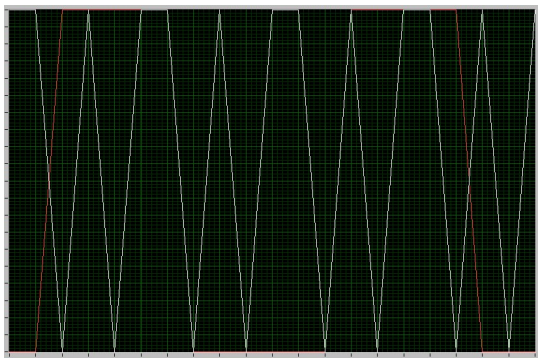

Figure 19. PLL Simulated Test Output

This output signal cannot be used as the square wave clock on which to base the rising edge triggers because it is not a good square wave. The edges of this signal are both inconsistent and poorly shaped. At high frequencies nearing 4 MHz, the output waveform alternated – sometimes randomly, and sometimes uniformly – between a square wave and a triangle wave. This is an unusually regular pattern in the output wave, which led us to believe that the test is failing not fully because of the code design and may have something to do with the computer that it was run on.

The software PLL also took around 60-90 seconds to establish the phase-stable output signal when tested with a lower frequency input signal at around 400 kHz. This long time makes the PLL unsuitable for the purpose of triggering measurements in its current state. There is a parameter of PLLs called “lock time”, which describes the length of time between the moment that the input signal is started and the moment that the PLL establishes the steady output signal “locked” to the input. Normally, this time is a few cycles or less. The whole minute of lock time we experienced during testing, as well as the performance of the software during the test led us to another limitation of the PLL software. The computer needs to have the necessary processing power to be able to keep up with the high frequency signal. We suspect the computer was the bottleneck during the tests. At low frequencies below 10 kHz, the PLL was very responsive, locking in only seconds. At higher frequencies, even after the PLL was established, the entire

computer was slow and unresponsive. Dependency on the machine's hardware specifications not only causes the lock time to be much longer, it also makes the lock time unpredictable and inconsistent between runs. While a long lock time can be worked around, the inconsistency is difficult to reconcile with programming or user procedures. Though the PLL has not been tested directly on the lab computer because of COVID-19, it is unlikely that the software could perform much better than the computer on which we ran the tests.

5.3 Data Processing

The processing testing is performed using a one-minute example TDMS file collected with photodetector and position channels. Some of the tests were also accomplished with synthetic data sets created in Excel, MATLAB, or Python. The main processing is written in Python 3 using the following libraries: `NumPy`, `sys`, `os`, `npTDMS`, `SciPy`, `math`, `sklearn`, `csv`, and `multiprocessing`. Some of the testing requires using `matplotlib`, which we use and recommend, for plotting the results to visually confirm the processing stage in question.

5.3.1 Individual Module Testing

The processing is started through the GUI using a command-line call to a Python executable. The command-line call will pass in the location, name of the input raw data file, relevant statistics from the measurement equipment and user selections, output filepath and filename, and arguments. An example call, which should be given all in one line, is shown below:

```
Python3 ProcessingCaller.py "InputPath" fourChannelMinute 128 128
"OutputPath" processedData 1 1 10 1 10 1 10
```

The command-line command will call `FreqClusterer` with 13 arguments, tabulated in Appendix C. `ProcessingCaller` starts the data processing and notifies the user by printing "Start getData" to the console as it opens the TDMS file. The screen capture below shows the test bench being run as "main" with the piece of processing under test as a function within the same document. The results of the test were printed to the right as shown in Figure 20 on the next page.

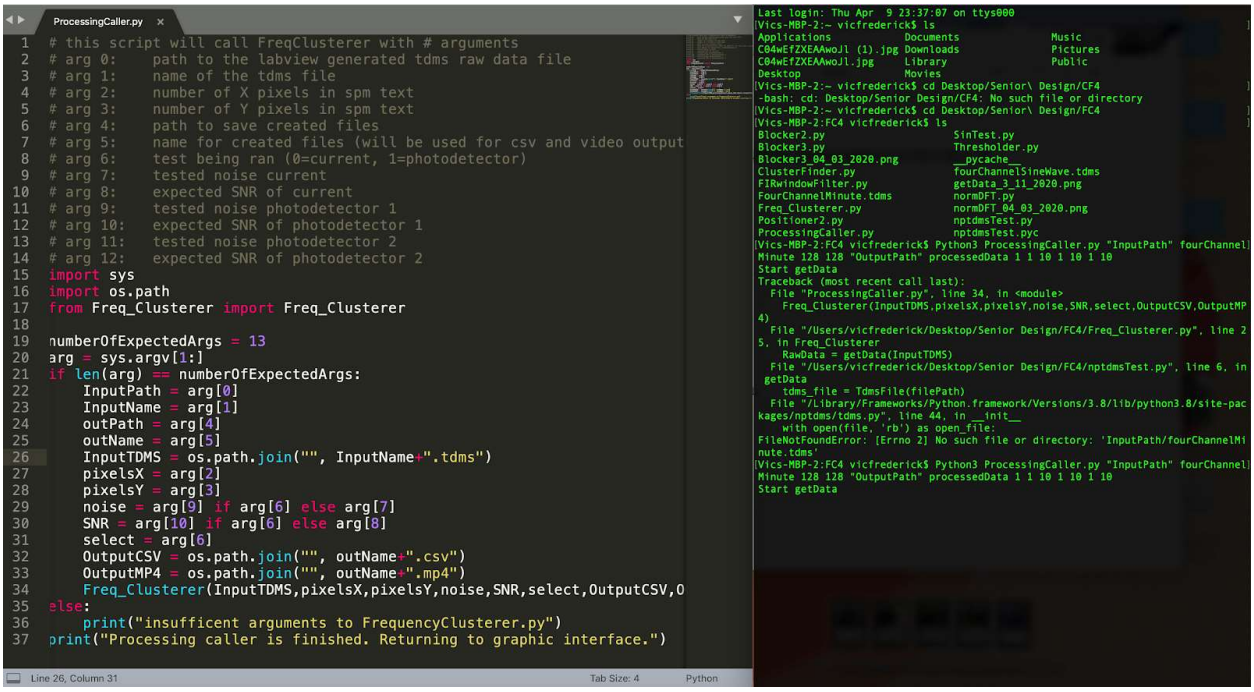

Figure 20. Processing Test 1: `ProcessingCaller`

To open the large amounts of data produced by the NI DAQ, our processing software uses the NI-native TDMS format, supported by the `npTDMS` library in Python, to open the file and extract each channel's information. The channels are stored together as a `NumPy`-type array of size 4xn samples. The testbench opens the designated TDMS file, stores it in a `np.array`, and reads out the shape of the created array, as seen in Figure 21.

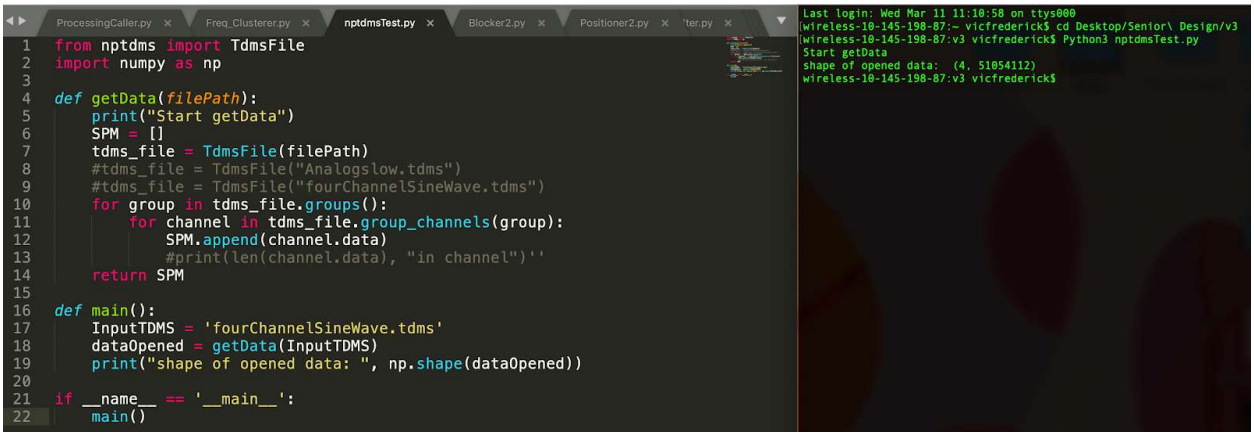

Figure 21. Processing Test 2: `getData`

The channel data is organized from the `np.array` as shown in Table 4. We want to organize the selected data channel according to the spatial and time location where it was taken. This organization, or blocking, is the first processing step.

Table 4. Channel Signals for SPM Tests

	Channel 0	Channel 1	Channel 2	Channel 3
Conductive Tests	Current Preamplifier	Not Connected	Tip Bias	X Position
Non-conductive Tests	Photodetector 1	Photodetector 2	Tip Bias	X Position

With a small sinusoidal containing an offset input, the testbench was run to export a selected XY-location as a plot. The large phase in the middle of the nonzero data arises from half of the data coming from the AFM's second pass across the location. Each location is zero-padded to maintain consistency for the later processing, such as the normalized fast Fourier transform shown in Figure 22.

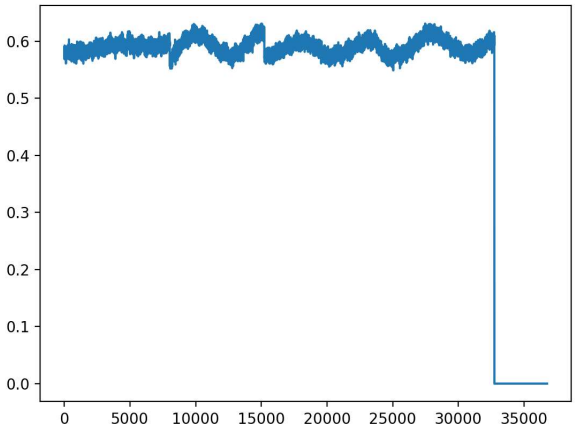

Figure 22. Processing Test 3: `Blocker3`

The Fourier transform implementation is in the `NumPy` Python library, but the normalization requires some validation. The normalization is done using the `normDFT` function shown in Figure 23. The testbench applies a sine wave with good period relevance to the number of samples for the FFT to pick out.

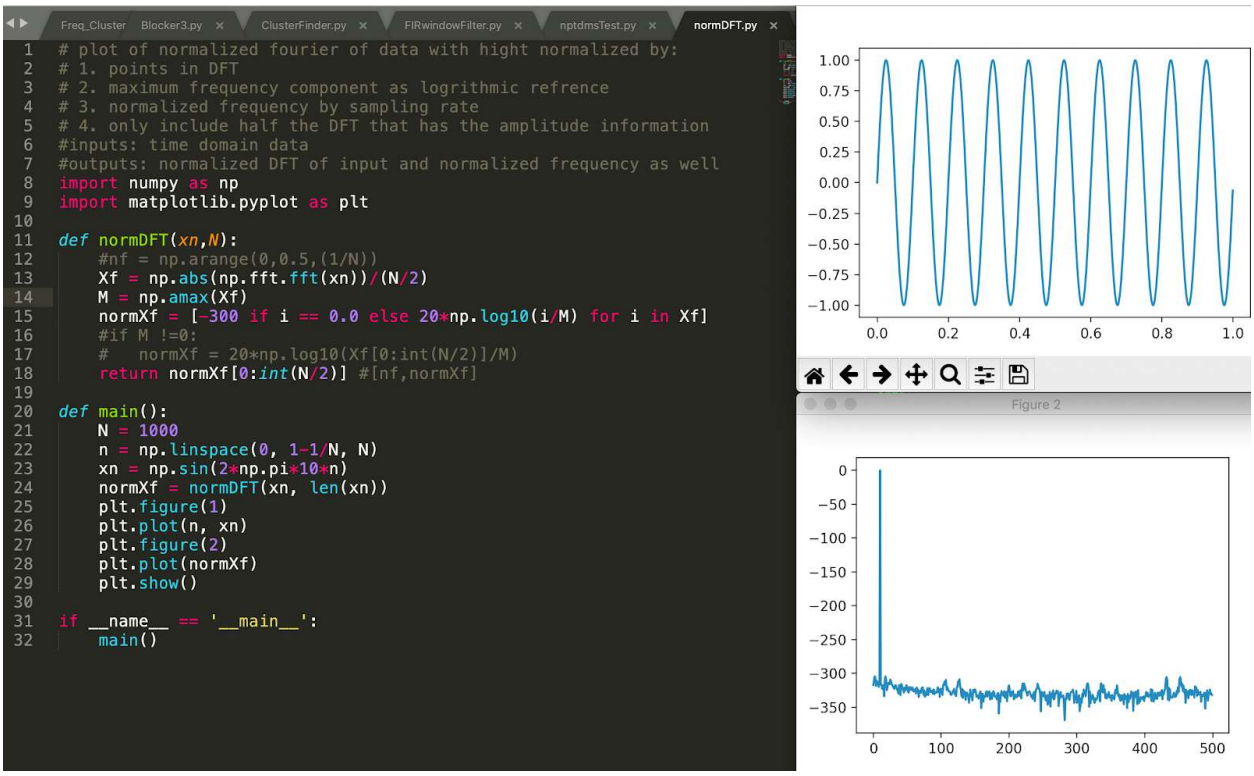

Figure 23. Processing Test 4: `normDFT`

In order to evaluate points across multiple Fourier transforms, each FFT was normalized such that projected signal power would be above a given negative decibel. The data must then be reorganized a second time for a more specially driven analysis if the points are above the threshold. The `thresholder` testbench in Figure 24 creates a random array of decibel values, evaluates the input statistics to generate a thresholder, and then restores the array with only the values above the threshold.

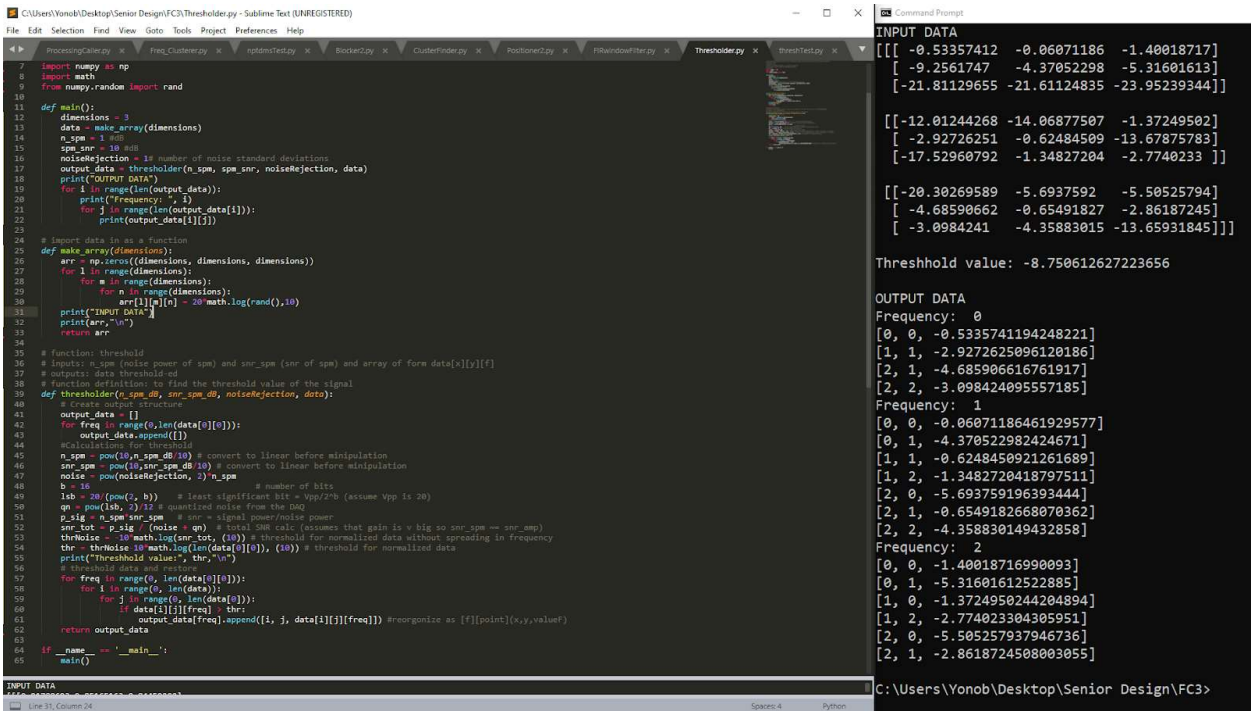

Figure 24. Processing Test 5: `thresholder`

Finding spatially dense high points of frequency responses is our primary processing goal based on the prior art and our stakeholder's interest. `ClusterFinder` picks out the centers of these highpoints, as shown in Figure 25. The testbench creates two sets of dummy centers with a random scatter around them in the figure below. After returning, the centers are exported to a CSV file.

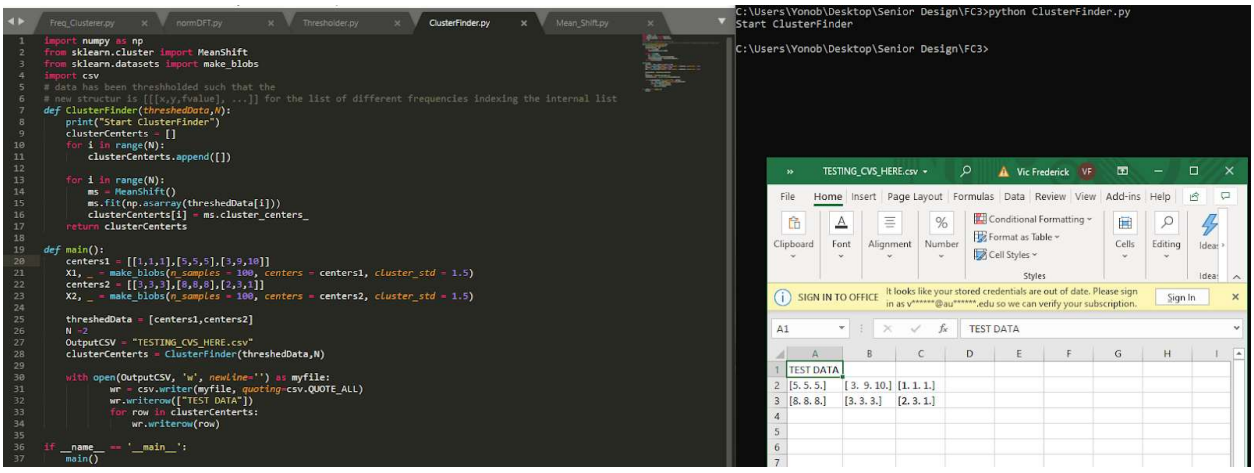

Figure 25. Processing Test 6: `ClusterFinder`

5.3.2 Integration

Because of the long run times of the data sets, most integration testing has been done with simulated data, but individual tests, as seen above, have largely been accomplished with both simulated and real data. With a simulated data set as small as a few points, Figure 26 shows how the data processing is capable of thresholding the raw data and extracting clusters of interest.

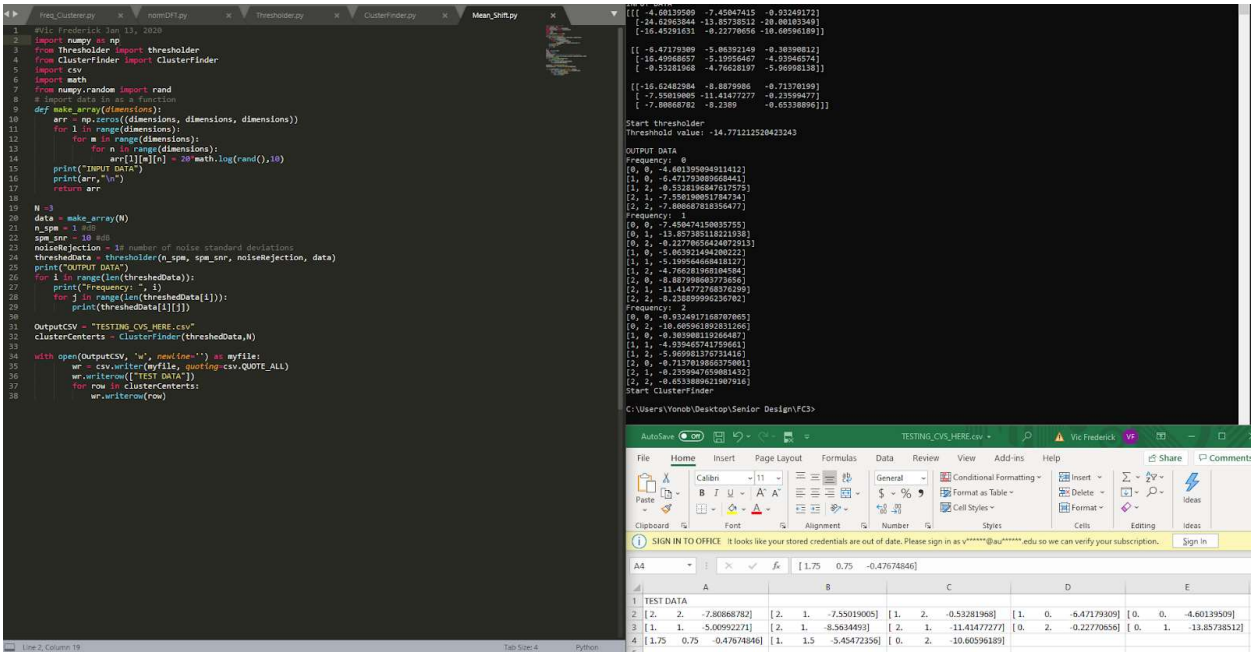

Figure 26. Processing Test 7: Integration

5.3.3 Visualization

After using artificial data to test the visualization code, the data from the DAQ and process integration were used to test the visualization software. We immediately noticed that the output data was in a different format than was readable by the visualization code, so this format had to be adjusted before testing was finished. The visualization will be called by the GUI when the user requests. Before integrating the GUI and the visualization, the command prompt was used to call and test the visualization. To run the 3D cluster visualization, the command is:

```
python visualizing_cluster.py data_file.csv frequencybin_number
```

where `data_file.csv` contains the datafile to be visualized and `frequencybin_number` indicates the frequency bin containing clusters of interest. Figure 27 shows a screen capture of the output of the 3D cluster visualization using the sample processed data as the first argument and choosing 400 as the frequency bin of interest.

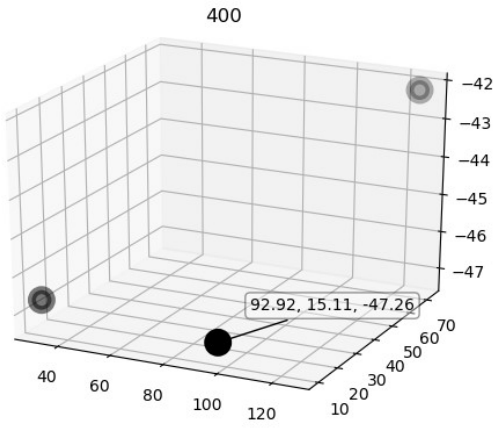

Figure 27. 3D Visualization at 400 Hz Frequency Bin

The visualization script that generates a movie was tested by running a command in the command prompt, although in reality the script will be called from the GUI with the command

```
python visualizing_movie.py data_file.csv movie_name.mp4
```

where `data_file.csv` is the data clusters to be visualized and `movie_name.mp4` is the name of the file the movie is being saved to. Figure 28 shows one frame from a generated colormap movie.

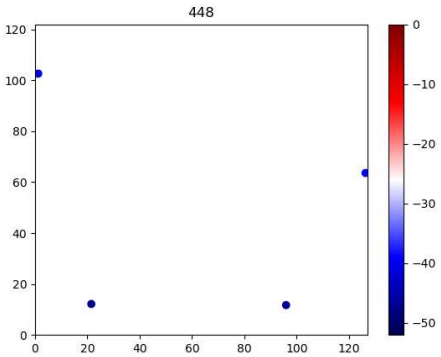

Figure 28. Screenshot of One Frame of Colormap Movie

5.4 GUI

Due to its simplicity in functionality, the GUI did not undergo formal testing. Instead, it was largely tested via manual inspection. To accomplish this, we simulated several common use cases for the GUI and, through inspection and logging, verified that the GUI behaved as expected in each case. Table 5 outlines these test cases and the expected behavior. Unfortunately, we were unable to test the GUI in a lab setting due to COVID-19, and testing of the GUI in the lab environment should be conducted when the lab reopens. However, we believe that the GUI should behave as expected in the lab as well.

Table 5. GUI Use Case Tests

Use Case	Expected Behavior
User clicks the "Start Test" button.	The LabVIEW exe for the DAQ is launched with the correct parameters, and the button turns red and reads "Stop Test."
User clicks the "Start Test" button when it reads "Stop Test."	The DAQ exe is successfully exited. Then, if enabled, the Python processing script is launched with the correct parameters. Finally, the movie generating script is launched upon processing completion.
User clicks the "Start Processing" button.	The processing software is launched with the correct parameters and exits with code 0.
User clicks the "Display Blob" button.	The blob visualization Python script is launched with the correct file and blob number. The visualization is displayed properly.
User clicks the "Generate Movie" button.	The movie generation Python script is launched with the correct file. The movie is successfully generated and opened. The mp4 file of the movie is saved to the specified location.

5.5 System-Level Testing

Following the individual subsystem tests, the integrated system should be tested thoroughly to ensure we are receiving accurate data and are processing it correctly. It is important that each subsystem documents their debugging so that when we test the system as a whole, we can more easily pinpoint the source of the issue based on comparing the problem we encounter to the previously documented problems. The unforeseen COVID-19 pandemic hindered our system integration testing by prohibiting our access to the laboratory where the SPM system is located.

We were unable to test the integration of the current preamplifier into the SPM system because of this.

In the future, when access to the lab and the SPM system is possible, the methods going forward include testing the system by first comparing the new system acquisition and processing to a test measurement from the current SPM setup. This can be done by acquiring a reference sample and measuring the test sample on the original SPM system, then measuring the sample again but using our system. Comparing the two datasets should give a general idea on whether the measurements are being acquired and processed accurately. It may also be helpful to consult prior art to find examples with which to compare our processing algorithms. We will then try to replicate the processes described in the prior art and compare the results. While there is no way for these results to completely confirm the accuracy of our measurements, these comparisons will allow us to run multiple trials and see if we are getting similar results to the prior art, which implies we are processing the data correctly as well.

6.0 TIME AND COST CONSIDERATIONS

Overall, the system developed mostly in line with planned time and cost projections. However, the COVID-19 outbreak severely delayed testing in most cases, causing significant time delays. Testing for the current preamplifier was one of the most delayed, and some extra costs were also incurred due to COVID-19. The DAQ system met cost projections but was delayed several weeks due to unforeseen errors in the controlling LabVIEW code. The data processing software had no cost considerations but was also delayed in testing because of the COVID-19 outbreak. The GUI software also had no cost considerations and was delayed because of necessary changes to the programming language choice.

6.1 Current Preamplifier

The current preamplifier, in many ways, developed in a circular fashion; in fact, the final design was one of the first ideas that we proposed but had scrapped at the time because we thought there was something better that we had not discovered yet. It was only through understanding all of the different gain, bandwidth, stability, and noise considerations that we could truly comprehend not only the key issues affecting preamplifier design, but also why and how the Ithaco preamplifier

was so limited in its bandwidth, and why the simple solution was the best. Since the Ithaco preamplifier was designed in the 1980s, it was limited by state-of-the-art amplifiers at the time. Our only real innovation was to replace the amplifiers in the Ithaco circuit topology with current state-of-the-art high-speed amplifiers that could handle 4 MHz signals. Even though the circuit turned out to be very simple and could have taken less than a month to get right had we just followed the Ithaco design, taking an entire school year to learn about and document all of the tradeoffs in amplifier design for the specific application of the SPM will be useful for future students to continue this work.

The current preamplifier was also the part of the project most affected by COVID-19. Due to the emergence of COVID-19 first in China, and then its spread over the world, we were unable to get functioning prototypes from either Chinese or US-based circuit board manufacturers before COVID-19 shut all the research labs. We made several pivots, first ordering from Chinese manufacturers, then from US-based manufacturers. This incurred additional costs that could have been saved if we could have predicted the eventual quarantine. By the time all the boards arrived, it was already spring break, and we did not have fabrication equipment at home. As a result, we solely relied on Multisim to test and evaluate the circuit.

6.2 DAQ

The DAQ system as a whole was plagued by extra time costs. The main system required much more debugging than the original schedule planned for. In addition, both the main system and the PLL were affected by having access to the hardware and the lab at Pickle Research Center closed by COVID-19.

6.2.1 Main System

The DAQ system met budget considerations but missed schedule considerations by about three weeks due to many errors in the LabVIEW code. Cost for the DAQ system matched what was expected since no extra parts were required beyond the initial purchase of the PXIe-6124 DAQ card, the PXIe-1071 DAQ, and the TB-2706 for wiring. Time constraints for the DAQ system were less accurate. Due to the unforeseen challenges of programming with LabVIEW, the software took much longer than expected to finish. Dr. Heath and NI engineers were very helpful

with resolving errors and debugging so that the system was finished only approximately three weeks behind the original schedule. The other issue with the DAQ system schedule is that exhaustive testing will not be able to take place at all by our team due to the Pickle Campus being shut because of COVID-19. However, the code has been tested in simulation and some testing was done involving using the function generator as previously mentioned. We expect that the DAQ system will function as intended when the Pickle Research Campus is reopened and available for use.

6.2.2 PLL

The PLL was initially considered a stretch goal and became viable in the first half of the spring semester, and it represented good progress in the other components of the project. However, because it was only started later in the design process, we did not complete it on schedule along with all of the other components. The time between the moment we began the PLL and the COVID-19 pandemic was not enough to work through the iterations of design and testing, especially because the team lost the ability to perform tests in the lab on the actual hardware. After the labs reopen, the design and testing process of the PLL can continue.

6.3 Data Processing

The data processing did not have any budget constraints since all the software applications were publicly available. While further, more complex analysis would have been advantageous to rigorously test the processing, the predetermined schedule to develop the processing algorithm and visualization was met. The visualization took a few weeks longer than intended due to overcoming the Python learning curve and a slow computer, however it was able to be completed before the final showcase. COVID-19 did not conflict with the design and development of the software due to software being a remote-friendly task; however, portions of the testing were not completed since we did not have access to the SPM system after the shelter in place for Austin was enacted.

6.4 GUI

The development of the GUI took about a month longer than intended, after its completion was delayed twice. First, due to the previously discussed bug that prevented the ReactJS-based GUI

from launching the LabVIEW executable and the Python scripts, the ReactJS-based GUI ultimately had to be scrapped, and a new GUI in JavaFX had to be built. Second, Katherine, the lead GUI developer, got sick while attempting to address the bug, which meant the start of the JavaFX-based GUI was delayed. However, the GUI was ultimately completed successfully in time for the project video to be produced.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

Several kinds of ethical and safety considerations apply to our design. For the current preamplifier, safety concerns included ways to limit injury to the user and components from current or voltage. For the DAQ system, there are concerns for error handling in the code and the specific power-on and power-off sequences so as not to mess up the installed driver software. The main safety concerns for the data processing software are documentation and outputting error messages to the user at correct times. The main ethical concern for the GUI software was to avoid plagiarizing existing code.

7.1 Current Preamplifier

In our design, we have several circuits that protect both the circuit and the user from sustaining injury. First, in the production schematic, we implemented a rocker switch to turn the device on and off. The rocker switch breaks the power circuit from the rest of the circuit when the device is not in use so that the sensitive op amp components will not be annihilated by a power surge. Bypass capacitors also ensure that the op amp power rails remain constant and resilient to changes in the power provided by the power circuit. Second, we chose to use a wall plug to convert AC electrical power from the grid to usable DC values. Coupled with an isolated DC-DC converter, the power circuit provides a safe, constant 9 V DC power supply to the op amps in the preamplifier. The 9 V supply sets the power rails of the op amps, so that they cannot maintain output voltages larger than 9 V.

There are several aspects of the current preamplifier that we did not have time to implement that could contribute to the safety of the device. First, there is no input overvoltage protection or current suppression. Spikes in SPM current may saturate the op amps and give unreliable readings. Large spikes may cause the transimpedance amplifier to try to output above its power

rating and burn out. Second, due to the sensitivity of the op amp components, the op amps may experience drifts in gain and response over the course of a year or two. The manufacturers recommend replacing the components or just buying a new board once a year to two years to maintain good response.

7.2 DAQ

As a system, the main safety concerns for the DAQ are related to the correctness of the software. Neither the software for the main system nor for the PLL implements any error checking or correction. Also, the driver software installed on the interfacing computer for the DAQ system requires a specific power-up sequence. The PLL needs considerations for the time it takes to start up.

7.2.1 Main System

The main safety concerns for the DAQ system are software related: the driver software installed in the lab computer requires the two parts of the system to be powered in a certain sequence, and error handling/checking is not yet implemented in the current version of the LabVIEW software. The first safety consideration for the DAQ system is that the two systems must be powered on and off in a specific order so as not to ruin the installed driver software: the DAQ must be powered on, and then the lab computer, and then the lab computer must be powered off before the DAQ is powered off. Essentially, the DAQ must be on the entire time the lab computer is on. The second safety consideration is error handling within the LabVIEW software in case a fatal error occurs during operation. Currently, if a fatal error arises, the LabVIEW program will simply crash. Due to complications introduced by COVID-19, error handling is not implemented in the current version of the LabVIEW software but could be added later when access to the SPM system is again available. The DAQ system is safe for users and best practices to prevent software errors are documented.

7.2.2 PLL

Special considerations for use must be taken into account because of the PLL lock time. Too long of a lock time means that the DAQ will not begin to take usable measurements until long after the SPM test begins. The system could deal with a longer lock time simply by starting the

SPM test before the measurements and discarding the data taken before accurate measurements began, even if the time is long, and the test could be repeated to collect any data missed the first time around. The more difficult problem comes from the inconsistent lock times. Even if the software is adjusted to only take data a minute after the PLL is started, some deviation in the input or in the parameters could cause the lock time to take even longer. Any data taken before the PLL is locked is inaccurate and should be discarded; however, it would be unfeasible for the user to know the difference between the good and bad data, so the test would either have to be repeated, or written off as unobtainable.

7.3 Data Processing

The data processing needed to consider the documentation of libraries used, assumptions made to ensure easy debugging and reliable information for future users, and outputting error messages which indicate when a projected error will occur. The safety concerns for the data processing are related to the version of Python and the Python libraries utilized. Mismatch in the Python and Python library versions can be prevented by documenting the versions currently used to write the code. Ethically, the biggest concern is to output as accurate as possible information from the data processing. Where assumptions were made, documentation should be prominently placed such that the user will understand how the clusters were developed and what they mean. By making this information readily available, the researcher operating the system will be able to more easily assess the validity and importance of the results they are receiving. The main safety precaution for the visualization is that there will be an error message displayed if a user inputs a frequency that is out of the range of the data set. This will prevent the user from having debug issues in the future.

7.4 GUI

The only ethical concerns we faced while developing the GUI is to avoid plagiarizing code and cite any sources we frequently referenced. We also designed font sizes to be large enough to be accessible to those with visual impairments. While other steps, such as designing stylesheets with contrasting colors, in the final GUI could have been taken and should be taken in the future, we decided to focus on testing the GUI's functionality with the rest of the system. In addition, other accessibility features that we considered but did not have time to implement include audio

buttons reading the on-screen text as well as interfacing with speech recognition software to make the GUI hands-free. These should be included in the next iteration of the GUI.

8.0 RECOMMENDATIONS

There are many future recommendations for our project, some because of plans that were pushed back due to COVID-19. For the current preamplifier, more testing must be done, and some extra circuitry could be implemented. For the DAQ system, more testing should be done, the LabVIEW code could be structured better, and more automation of tasks could be added. For the data processing code many more libraries and extra kinds of processing can be integrated. For the GUI, more user features can be added for safety and ease-of-use.

8.1 Current Preamplifier

Future work on the current preamplifier, besides ordering boards and testing, should implement overvoltage and current suppression circuitry to prevent large spikes in SPM current from destroying sensitive op amp components. Some suggestions on how to approach overvoltage protection and current suppression are given in the Ithaco preamplifier manual. In addition, future designers should investigate higher-order and active filters to continue to decrease the noise in the preamplifier output and re-optimize the current filters based on data from an actual hardware printed circuit board. Finally, future users should consider building a metal chassis and enclosure for the device to keep electromagnetic interference from entering the circuit. Alternative wiring solutions may also improve input and output signal quality, which will be useful for better data processing.

8.2 DAQ

For the DAQ system in the future, more testing should be done on both the main system and the PLL. Also, the PLL code should be integrated with the main system code, which will then require other tests to ensure correctness and reliability. For the main system code, more modularity and user configuration could be achieved. Also, it would be nice to design an automated solution to switch signals on analog input channel 0. For the PLL, a hardware solution might need to be designed to replace the current software solution.

8.2.1 Main System

In the future, another category of tests should be done on the DAQ system, the code can be made more modular and allow for more runtime customization, and an automated system can be designed to make switching signals on analog input channel 0 easier. As mentioned previously, one more kind of test involving sampling on known samples with the SPM system is recommended. This can be completed once access to the Pickle Research Campus lab is allowed. It would also be a nice feature for users and anyone maintaining the LabVIEW code for the LabVIEW code to be more modular, and possibly for more settings to be customizable at run time by the user through command line arguments from the GUI. In addition, a system to automate the switching of the analog input channel 0 wiring from the tip bias signal to the vertical photodiode signal would be very useful. Mainly, future work on the DAQ system includes one more kind of test, a more customizable user experience, and an automated solution for switching the input signal on analog input channel 0.

8.2.2 PLL

In the future, the PLL code should be modified to more efficiently create a stable square wave to output on one of the DAQ's digital output lines. Once there is a sharp enough rising edge in the system, the signal can be fed back into an input to be used as a sampling trigger. Some more code would have to be written to use this sampling trigger signal as the clock to synchronize DAQ sampling. After the lab is reopened, a true hardware test can be run where the PLL output is confirmed to be a sufficiently accurate and consistent signal relative to the input from the amplifier. This can be verified with an oscilloscope connected to both the tip signal and the PLL output signal. It would also be good to take into account the possibility that the desired functionality of the PLL as designed is not achievable in the system, either because of software design or, more likely, the technical specifications of our computer or DAQ hardware. In the case that the desired functionality is impossible, the responsiveness of the software PLL with all its user and software adjustability might be sacrificed for a consistent but rigid hardware option. Designing the PLL as a hardware component alongside the preamplifier could give the reliable 4 MHz square wave that we need, at the cost of the ability to easily configure the PLL in software.

8.3 Data Processing

The recommendations for the data processing modules include expanding the processing algorithm to include other types of measurements, integration of machine learning to determine interesting patterns in frequency response over time, updated visualization features, and continuation of the multiprocessing integration. Going forward there are numerous other methods that can be integrated into our system to gather interesting data from various other types of measurements, for example capacitance and surface charge information. There are a number of experiments that can be run on the SPM system, such as Kelvin Probe Microscopy. In addition, it could be interesting to integrate further machine learning aspects into the data processing. If the system could gather the processed data and find patterns in the frequency response that change with time, for instance, then that eases the burden on the user. Instead of watching the video and finding patterns that relate to each cluster as frequency changes, the machine will be able to find and pinpoint these interesting patterns. In addition, as these new algorithms are developed, new scripts will need to be integrated in order to visualize the data. One interesting idea is to have the functionality of hovering over a cluster's amplitude on the colorbar and developing a movie that will play showing all the frequency clusters at the chosen frequency bin. Playing a movie that shows specific frequency clusters would help the user find frequency bins with similar characteristics. Finally, it would also be helpful to put more research and time into integrating the multiprocessing more fully into the data processing code. Restructuring the code to allow for much more multiprocessing would allow significant speed up to be achieved even on the largest datasets, particularly since many of the data processing algorithms are made up mostly of matrix manipulation and multiple indices can be modified at the same time without interfering with each other.

8.4 GUI

As new features--namely, the preamplifier and the PLL--are incorporated into the system as a whole, new features will need to be added to the GUI. First, several warnings should be added to protect the preamplifier when it is incorporated into the system. These warnings include:

- A warning that appears when the preamplifier is enabled that tells the user to turn the preamplifier off if not using it for current measurements.
- A warning that currents greater than 10mA may damage the amplifier board.

- A warning that appears when “10pA to 2nA” or “10pA to 200pA” are selected that notifies the user that noise may obscure true measurements for currents these small.
- A warning that indicates the bandwidth of the selected amplifier gains and tells the user not to sample over this frequency.

Further, after the PLL is incorporated into the system, a countdown timer that tells the user when the DAQ has been initialized and is ready to begin collecting data is recommended. Because the PLL can take several minutes to initialize, and the DAQ cannot begin collecting data until after the PLL finishes initializing; a fixed time greater than the PLL initialization time can be chosen for this countdown. After the timer has finished, the GUI can indicate to the user that it is safe to start the SPM. Finally, designing for accessibility, discussed in Section 7.4, should be included as a priority in later iterations of the GUI.

9.0 CONCLUSION

During the course of this project, our team aimed to design, construct, and demonstrate a high-speed data acquisition and processing interface that interacts with a scanning probe microscope to generate nanoscale images for analysis. We modified an existing SPM system to collect data on the order of megahertz and generate time series to achieve highly dynamic and sensitive readings at the nanoscale. These modifications included adding a current preamplifier, an external DAQ, data processing and visualization modules, and a GUI. While we were able to design prototypes for each of these subsystems, we were unfortunately unable to build and thoroughly test all of them due to COVID-19. As a result, there is much work we recommend be done on the project in the future.

First, our system was not integrated and rigorously tested, as we had set out to do at the beginning of the year. Each subsystem was designed, implemented, and tested to the best of our abilities, given the COVID-19 crisis. However, while we were able to show our preamplifier design worked in simulations, a completed preamplifier board was never assembled. Furthermore, the PLL was never tested within the DAQ software in a lab setting. We were also unable to integrate our subsystems into a complete system, due to the lab being closed because of COVID-19. As a result, we were unable to do system-level testing on our system, and we have to

rely on the results of our subsystem-level testing to prove our implementation will work as expected.

In the future, there is much work to still be done on the project to complete each subsystem and ultimately integrate the system. First, the preamplifier prototypes need to be assembled and tested. Next, the PLL needs to be integrated into the DAQ code, and the DAQ subsystem needs to be further tested in the lab environment. The multiprocessing work needs to continue on the processing software to make it more efficient, and the processing software should be tested on more data sets. As the PLL and the preamplifier are integrated into the system, new warnings and popups need to be added to the GUI for safety purposes. Finally, the entire system needs to be assembled in the lab and rigorously tested.

All in all, given the unexpected complexity of our project, the unforeseen technical challenges we encountered, and the unprecedented circumstances caused by COVID-19, our team is extraordinarily proud of the work we have been able to accomplish this semester. We are thankful for the opportunity to work on such a project, and we greatly appreciate all the resources and support we received along the way. In the future, we hope that other teams will continue and build upon the work we have done, and our system will one day be used to further nanoscale research.

ACKNOWLEDGEMENTS

Over the course of two semesters, we encountered many people in our lives who supported us, aided us, counseled us, and advised us as we progressed in our project. We would like to thank Dr. Edward T. Yu for proposing and initiating the senior design project in the first place and guiding us through the project even when there was no clear sight of the end. We would also like to thank Dr. Yu's graduate students, particularly Ted Kim, for mentoring us in our initial exposures to the SPM, training us, and collecting all the usable data for us because we were too scared put a really expensive machine out of commission for six months. In addition to Dr. Yu and his research assistants, our technical teaching assistants, Javier Rodriguez-Fernandez and Kassandra Perez, and our year-long writing teaching assistant, Hanan Hashem, formed the backbone of our support network in terms of the yearlong senior design course and provided valuable feedback to the project and the paper. We would also like to thank Dr. Robert Heath and Dr. Bill Fagelson for providing a framework and structure to approach a truly massive intellectual and technical endeavor. Along the way, we also had many valuable conversations with National Instruments engineers, Dr. Brian Evans, and Dr. Vijay Garg, who aided in various aspects of the project. Mark Innmon, technical staff in the ECE labs, also provided tremendous support through the soldering lab and his many years' wisdom and experience soldering pesky integrated circuit chips with and without a reflow oven. Lastly, and perhaps most importantly, we cannot forget all the cats, memes, and cat memes that helped us remain at least somewhat sane throughout the senior design course and the course of the project. God bless the internet.

REFERENCES

[1] Solares D. Santiago, "Subsurface Imaging of Soft Matter by AFM", Imaging & Microscopy, Mar. 16, 2015. [Available Online]. https://www.imaging-git.com/science/scanning-probe-microscopy/subsurface-imaging-soft-matter-afm

[2] "NI 6124/6154 User Manual", PXIe-6124 Refrence Material, National Instruments. Aug. 2018. [Available Online]. http://www.ni.com/pdf/manuals/372613a.pdf

[3] *Model 1211 Current Preamplifier,* DL Instruments, Ithaca, NY, USA, 2000.

[4] H. Hashemi, "Transimpedance Amplifiers (TIA): Choosing the Best Amplifier for the Job," Texas Instruments Incorporated, Dallas, TX, Application Report SNOA942A, Nov. 2015.

[5] X. Ramus, "Transimpedance Considerations for High-Speed Amplifiers," Texas Instruments Incorporated, Dallas, TX, Application Report SBOA122, Nov. 2009.

[6] L. Orozco, "Programmable-Gain Transimpedance Amplifiers Maximize Dynamic Range in Spectroscopy Systems," *Analog Dialogue,* vol. 47, no. 5, pp. 1-5, May 2013.

[7] "PLL - Phased Locked Loop," *NI Community*, National Instruments 30-Jan-2017. [Available Online]. https://forums.ni.com/t5/Example-Code/PLL-Phased-Locked-Loop/ta-p/3492731?profile.language=en

[8] L. Collins, A. Belianinov, S. Somnath, N. Balke, S. V. Kalinin, and S. Jesse, "Full Data Acquisition in Kelvin Probe Force Microscopy: Mapping Dynamic Electric Phenomena in Real Space Scientific Reports", Oak Ridge National Laboratory. 12 August 2016.

APPENDIX A – SUPPLEMENTAL INFORMATION FOR CURRENT PREAMPLIFIER

APPENDIX A – SUPPLEMENTAL INFORMATION FOR CURRENT PREAMPLIFIER

Table A-1. Annotated Bibliography

[1]	H. Hashemi, "Transimpedance Amplifiers (TIA): Choosing the Best Amplifier for the Job," Texas Instruments Incorporated, Dallas, TX, Application Report SNOA942A, Nov. 2015. This application report from Texas Instruments contains a good high-level overview of how to choose an op amp to use in a TIA circuit and gives an example of the decision process with three of TI's own amplifiers. It also discusses post-TIA amplification in greater detail. Most of the theoretical content in this user guide (and the final preamp design) was developed from this article.
[2]	X. Ramus, "Transimpedance Considerations for High-Speed Amplifiers," Texas Instruments Incorporated, Dallas, TX, Application Report SBOA122, Nov. 2009. This source is similar to [1], except that it takes a slightly more mathematical approach to choosing a good op amp for a TIA circuit. It is also a good summary of key considerations for TIA design.
[3]	P. C. D. Hobbs, "Photodiode Front Ends: The REAL Story," *Optics & Photonics News,* vol. 12, no. 4, pp. 44-47, Apr. 2001. This article gives various ideas and circuit topologies for increasing bandwidth using non-traditional circuit topologies. The author discusses bootstrapping, cascodes, and other non-linear devices that may improve bandwidth and SNR. It may provide other ideas to increase bandwidth without sacrificing gain.
[4]	*Model 1211 Current Preamplifier,* DL Instruments, Ithaca, NY, USA, 2000. This manual is the original Ithaco current preamplifier in the lab. It contains a block diagram of the preamp, diagrams for most of the internal circuitry, a bill of materials, and more on the device, but not a lot on the theory of operation. Another datasheet lists the main features of the Ithaco preamp, but does not have as much detail.
[5]	J. Ardizzoni, "A Practical Guide to High-Speed PCB Layout", *Analog Dialogue,* vol. 39, no. 9, Sept 2005. This article gives many practical considerations for high-speed PCB layout. For high-speed amplifier circuits, bypass capacitors on power rails and short trace lengths (smaller distance for signal to travel) were the main takeaways from this article. Other ideas, such as using surface-mount (SMD/SMT) components, ground planes, and vias were also useful.

[6]	D. Kleijer, *Op-amp noise calculator.* Accessed on: Apr. 20, 2020. [Online]. Available: http://dicks-website.eu/noisecalculator/index.html This website is an op amp noise calculator. It gives various common op amp topologies and lists the formulas for the noise contribution of each component. The site does not include a transimpedance topology, but it does include a non-inverting and inverting op amp. It is most useful for calculating the noise figure of the post-amplifier, which can then be used to calculate the resulting SNR of the entire current preamplifier with Friis' Formula.
[7]	M. Steffes, "Noise Analysis in High-Speed Op Amps," Texas Instruments Incorporated, Dallas, TX, Application Report SBOA066A, Oct. 1996. This application report from TI gives a comprehensive overview of noise analysis for high-speed operational amplifiers. It contains much of the theory and modeling techniques used to calculate spectral noise density and integrated noise.
[8]	J. Karki, "Active Low-Pass Filter Design," Texas Instruments Incorporated, Dallas, TX, Application Report SLOA049B, Sep. 2002. This application report from TI gives several suggestions for active low-pass filters. It goes into the theory and proposes several types and topologies for higher-order low-pass filters, including the Sallen-Key and multiple feedback (MFB) architectures. Higher-order filters may be able to reduce current preamplifier noise to below DAQ-detectable levels.
[9]	L. Orozco, "Programmable-Gain Transimpedance Amplifiers Maximize Dynamic Range in Spectroscopy Systems," *Analog Dialogue,* vol. 47, no. 5, pp. 1-5, May 2013. This application report from Analog Devices outlines the theory of operation behind the gain-setting circuit used in the current preamplifier design. The article also notes that parasitic switch capacitances can reduce the bandwidth of the current preamplifier and proposes different solutions to reduce the effect of parasitic capacitances in the analog multiplexers.

Table A-2. Transimpedance Amplifier Noise Phenomena

Type of Noise	Definition	Source	Effect
Input offset voltage	Voltage difference between input terminals of op amp	Op amp transistor non-idealities	DC offset voltage on output; can be nulled with circuits
Input offset current	Current difference between input terminals of op amp	Op amp transistor non-idealities	Generally not important for TIAs
Input bias voltage	Voltage necessary to bias internal op amp transistors	Op amp transistor non-idealities	Generally not important for TIAs
Input bias current	Current necessary to bias the internal op amp transistors, flows into terminals of op amp	Op amp transistor non-idealities	Some of the input current signal has to go into op amp, so less current going through feedback path, which causes output error
Input voltage noise	Small voltage fluctuations at input terminals (Flicker aka 1/f noise at low freq; shot noise at high freq)	Op amp transistor non-idealities	Disproportionately affects frequencies less than 100 Hz; distorted AC response
Input current noise	Small current fluctuations at input terminals	Op amp transistor non-idealities	Disproportionately affects frequencies less than 100 Hz; distorted AC response
Output voltage noise	Small voltage fluctuations at output terminal	Op amp transistor non-idealities	Usually not dominant source of noise, eclipsed by other noise sources
Thermal noise	Thermal excitation of charge carriers in resistors	Feedback resistor	R_f is inversely proportional to thermal noise
Input Capacitance	Capacitance on the input terminals	Op amp terminals, PCB capacitance, input wire (e.g. BNC) capacitance	Reduces bandwidth

Table A-3. Input Current Range Mappings

Input Current Range	Arg 1	Arg 2	Arg 3	Arg 4
0.1 mA to 10 mA	+0	+0	+0	+0
10 uA to 1 mA	+0	+0	+255	+0
1 uA to 100 uA	+0	+255	+0	+0
100 nA to 10 uA	+0	+255	+255	+0
10 nA to 1 uA	+255	+0	+0	+0
5 nA to 100 nA	+255	+0	+0	+255
0.5 nA to 10 nA	+255	+0	+255	+255
50 pA to 1 nA	+255	+255	+0	+255
10 pA to 100 pA	+255	+255	+255	+255

APPENDIX B – DAQ PINOUT TABLE

APPENDIX B – DAQ PINOUT TABLE

	Pin Name	Pin Number	Signal	Connected To
ANALOG IN	AI0 +	68	Tip Bias OR Vertical Photodiode	Current Preamplifier OR Signal Access Module
	AI0 -	34	Vertical Photodiode	Signal Access Module
	AI0 GND	67	Tip Bias	Current Preamplifier
	AI1 +	33	Lateral Photodiode	Signal Access Module
	AI1 -	66	Lateral Photodiode	Signal Access Module
	AI2 +	65	Tip Voltage	Signal Access Module
	AI2 -	31	Tip Voltage	Signal Access Module
	AI3 +	30	X Tip Position	Signal Access Module
	AI3 -	63	X Tip Position	Signal Access Module
DIGITAL OUT	P2.0	37	Mux Select 2	Current Preamplifier
	DGND	36	Mux Select 2 GND	Current Preamplifier
	P2.4	2	Mux Select 0	Current Preamplifier
	DGND	35	Mux Select 0 GND	Current Preamplifier
	P2.6	1	Mux Select 1	Current Preamplifier
	DGND	35	Mux Select 1 GND	Current Preamplifier
	P2.7	39	Mux Select 3	Current Preamplifier
	DGND	36	Mux Select 3 GND	Current Preamplifier

APPENDIX C – PROCESSING MODULE ARGUMENTS

APPENDIX C – PROCESSING MODULE ARGUMENTS

Argument Number	Description
0	Path to the labview generated TDMS raw data file
1	Name of the TDMS file
2	Number of X pixels in spm text
3	Number of Y pixels in spm text
4	Path to save created files
5	Name for created files (will be used for CSV and video outputs)
6	Test being ran (0=current, 1=photodetector)
7	Tested noise current
8	Expected SNR of current
9	Tested noise photodetector 1
10	Expected SNR of photodetector 1
11	Tested noise photodetector 2
12	Expected SNR of photodetector 2