Vibration Mitigation for Atomic-Resolution Imaging

Jonathan Goettsch

S.B. Degree Candidate in Mechanical Engineering

Harvard John A. Paulson School of Engineering and Applied Sciences Harvard University

Faculty Advisor

Jennifer E. Hoffman

In partial fulfillment of the requirements for the degree of

Bachelor of Science at Harvard University

Cambridge, MA

May 11, 2020

Acknowledgements

I would like to thank my faculty advisor, Professor Jennifer Hoffman, for guiding and supporting me throughout this project. I would like to thank Harris Pirie, a graduate student in the Hoffman Lab, for helping me complete this project, the work would have paled in comparison with him. I would also like to acknowledge Albert Chen, a former ES100 student, who built an incredible foundation to build from and Bryce Primavera, a former student in the Hoffman Lab, who pioneered the basis of this project. I would like to thank Linsey Moyer, Nishant Sule, and all the ES100 teaching staff for their guidance and help that made this a complete senior project. I would finally like to thank my family and friends for their help during this time.

In dedication to my family, coaches, and friends for their love and support throughout my college career.

Abstract

One of the biggest technical challenges facing scanning tunneling microscopes (STM) is limiting external vibrations. This challenge arises from the extreme sensitivity required to measure the atomic and electronic structure of materials. To reduce environmental vibrations, STM labs use passive isolation systems such as custom anechoic chambers, pneumatic isolators, and massive inertial blocks, making these measurements possible. The Hoffman Lab, with whom this project was completed, can successfully reduce vibrations to the several-picometer scale. However, residual coherent vibrations still inhibit their STM performance. This thesis presents a post-processing algorithm and measurement methodology that can reduce these coherent environmental vibrations through calibration of the system's transfer function and then the subsequent cancellation of estimated vibrations. Before the primary material measurements, a set of coefficients are calibrated to characterize the propagation of vibrations detected by a geophone to the STM tip. Vibrations are also recorded by the geophone during the material measurements. The vibrational noise experienced by the STM tip is then removed using the geophone signal, processed with the calibrated coefficients. An increase in the signal-to-noise ratio achieved with this algorithm effectively reduces the time needed to take measurements, increases the size of images that can be taken, and increases the resolution of the data. This system can also be applied retroactively to any STM system and is useful to anyone needing to remove vibrational noise after data has been collected.

Contents

Introduction 10

1.1 Background 10

1.2 Problem Definition 13

1.3 State of the Art 13

1.4 Targeted Users 16

1.5 Design Specifications 17

System Design 20

2.1 Theoretical Background 20

2.2 System Implementation 29

Topographic Noise Cancellation 32

3.1 Coefficient Calibration 32

3.2 Effective Noise Reduction 37

Spectroscopic Noise Cancellation 40

4.1 Effective Noise Reduction 41

Conclusions 45

5.1 Project Success 46

5.2 Future Work 47

Spectroscopic data cancellation 48

Synthetic Data Creation 52

Data parsing 53

1. Frequency chirp isolation 53

2. Sweep change identification 55

Python Code 58

Budget: 89

Works Cited 90

List of Figures

Figure 1.1: Two methods of STM operation are at a constant z-axis position (a) or being held at a constant current by a feedback loop (b). 11

Figure 1.2: Passive isolation systems in the Hoffman lab STM reduce environmental vibrations to the picometer scale. 12

Figure 1.3: Frequency spectrum of vibrations detected in Hoffman lab under ambient testing conditions. 14

Figure 1.4: Active cancellation techniques interfere with the STM measurement itself to remove vibrations 15

Table 1.1: Technical Specifications 19

Figure 2.1: Kappa is experimentally calculated. 28

Figure 2.2. System architecture showing the steps data processing taking between Nanonis, LabVIEW, and Python. 30

Figure 2.3. Detecting and eliminating tip change, and thermal drift 31

Figure 3.1. Calibrating the harmonic response of the STM with a speaker. 33

Figure 3.2: Calibrated coefficient matrix. 35

Figure 3.3: Calibrated coefficients in frequency domain. 36

Figure 3.4: Comparison of the fundamental coefficient and vector transfer function. 36

Figure 3.5. A 52.6% RMS value reduction in noise between the measured and matrix processed Z trace. 38

Figure 3.6: The matrix transfer function adds harmonics to the signal instead of removing them. 39

Figure 3.7: The magnitude and phase of noise in the raw data is not accurately estimated. 40

Figure 4.1: RMS value of current during a voltage sweep reduced by 32.6% and the corresponding I/V curve using Equation 2.19 (Method 1). 42

Figure 4.3: Reduction in the standard deviation from the average swept value indicates reduction in vibrational noise...... 43

Figure 4.4: Driving frequency reduced in LIY data. 44

Figure 4.5: Low frequency vibrations are shifted about the modulation frequency produced by the lockin amplifier.. 45

Figure A.1 High amplitude situations do not allow a linear approximation of the exponential relationship between tip position and current. 50

Figure B.1- The synthetic data represents previously collected data that can be generated and manipulated to test the abilities of the cancellation algorithm...... 53

Figure C.1.1: Determining the beginning and end of the calibration data set. 54

Figure C.2.1: Sectioning spectroscopic data with the voltage sweep data. 56

Figure C.2.2: Sectioning each step in the voltage sweep...... 57

List of Tables

Table 1.1: Technical Specifications .....**Error! Bookmark not defined.**

Chapter 1

Introduction

1.1 Background

Imaging the atomic structure of materials poses considerable challenges, given the size of atoms. Yet, successful atomic-resolution images have revolutionized our understanding of quantum materials, driving numerous and profound discoveries. A notable example of this is the creation of the quantum corral by which IBM have tried making atomic-scale electron circuits [1] [2]. A scanning tunneling microscope (STM) produces atomic-resolution images by moving a conductive tip across a metallic surface while measuring the tunneling current. The key to achieving atomic resolution with this microscope lies in the successful mitigation of vibrations transferred to the system from the environment. Quantum tunneling requires the distance from the tip to the sample surface to be between 4-7 angstroms [3]. At that range, electrons tunnel through the vacuum between the tip and sample, creating a current that depends exponentially on their separation [4]. This thesis will focus on two different operating modes of the STM (see Figure 1.1). The first mode collects topographic data by recording the tip's height trace as it moves across the surface, while holding a constant voltage across the junction. In this mode, a PI (proportional-integral) controller keeps the tunneling current constant by adjusting a piezoelectric actuator to change the Z position of the tip. The second mode

collects spectroscopic data, which reflects the electronic properties of the sample. This data is collected by recording the current while the tip is in a fixed position, with the feedback system off, while sweeping and modulating the voltage bias with a lock-in amplifier. The rate at which the current changes with the changing voltage (*dI/dV*) is proportional to the material's electronic density of states. This can distinguish a range of quantum materials, including superconductors, topological insulators, and many others [5].

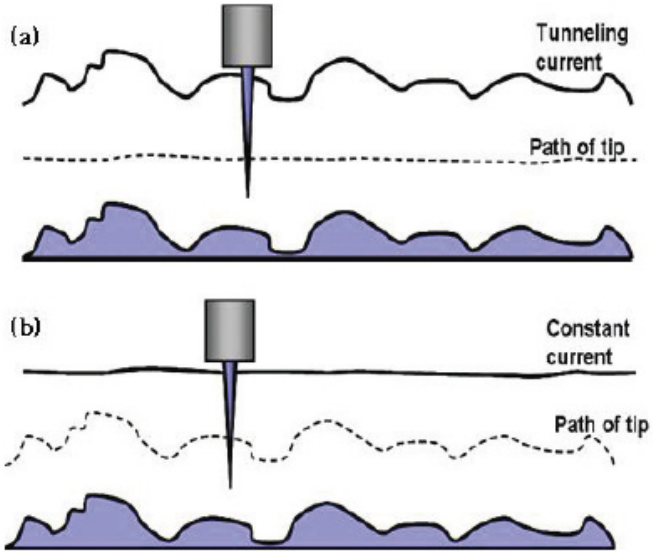


Figure 1.1: Two methods of STM operation are at a constant z-axis position (a) or being held at a constant current by a feedback loop (b). When a surface is scanned using method (a) the atomic structure is characterized. Sweeping the voltage while measuring with method (b) characterizes the electronic structure of the material *[3]*.

Modern STM labs use a combination of methods to passively limit vibrations. For example, in the Hoffman lab, pneumatic or mechanical springs are used to float a 20-tonne inertial block to isolate the system from building noise and reduce the resonance of the system to ~1 Hz [6]. In addition, the STM is

placed inside an anechoic chamber to prevent sound reverberation. Inside the room, the microscope is further isolated by placing it on a secondary inertial table that floats on independent air springs (see Figure 1.2). Despite these extraordinary lengths, tip-to-sample vibrations of 1-5 picometers are still common and limit the experimental capabilities. In this project, I implemented a new approach to cancel these residual picometer vibrations through a post-processing algorithm, expanding on work done by a previous ES100 student [7]. The advantage of a post-processing solution is that it can be used in any system, even well-isolated ones like the Hoffman Lab. Moreover, the approach is modular as it does not affect the raw data collected, allowing improvements through subsequent software updates.

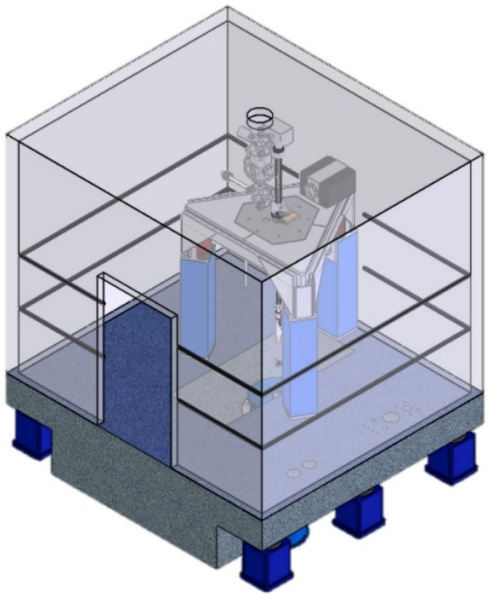


Figure 1.2: Passive isolation systems in the Hoffman lab STM reduce environmental vibrations to the picometer scale. These custom isolation systems cost labs millions of dollars to reach the ultra-low levels of vibration *[6]*.

1.2 Problem Definition

Environmental vibrations negatively affect the quality of STM data. STMs can have a depth resolution on the picometer scale and due to the exponential relationship with tip position (*Z(t)*), $I \propto e^{-\kappa Z}$, current data (*I(t)*) is severely affected by unintentional changes in position. Increasing the signal-to-noise ratio (SNR) of the data will allow the researchers to collect data more quickly, and not have to rely as heavily on averaging to reach the same SNR. This improvement will enable new measurements that are currently prohibited by the limited hold time, about 2–7 days, of the STM at low temperatures (due to the need to refill the liquid helium dewar periodically). This project aims to reduce the amount of vibrational noise in STM data, increasing its SNR past what has already been achieved by passive isolation systems and in past work.

1.3 State of the Art

A large amount of work has been put into characterizing and mitigating vibrations incident upon STMs. Due to the nature of its operation, every STM must account for vibrations to some degree. M. Hamidian simplified the STM as a two-stage mass-spring-damper system. He characterized the frequency response of the system through variations of the inertial masses, spring constants, and damping constants to determine their optimal related values [6].

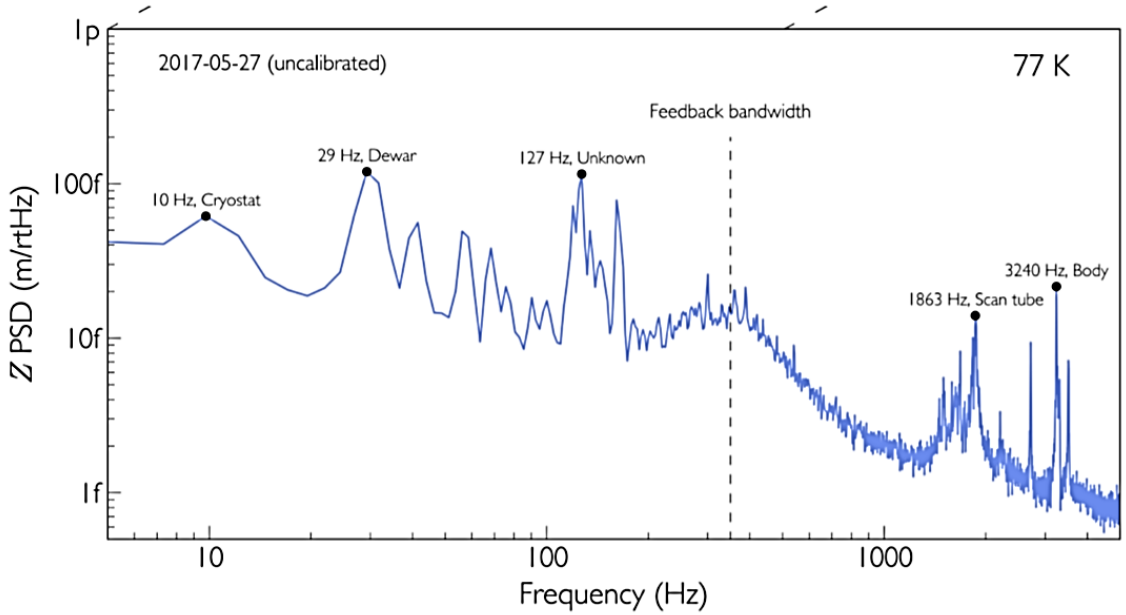
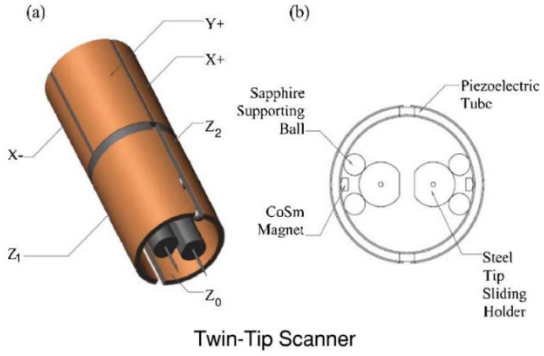


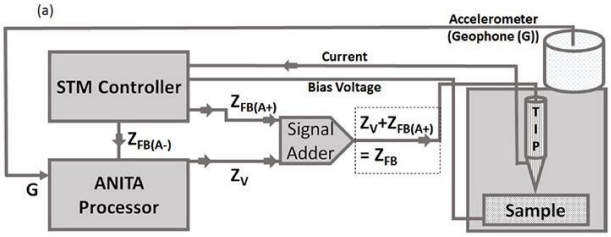
Figure 1.3: Frequency spectrum of vibrations detected in Hoffman lab under ambient testing conditions. With every passive isolation system active, the STM can measure ambient noise. Sources of this coherent noise are found based on their resonant frequency. This level of noise can hide smaller surface features of materials that are less than 2 pm. *Copied from ref [8] [9]*

Olivia et al. performed a similar analysis, instead modelling the system as a three-stage mass-spring-damper system [10]. Iwaya et al. utilized both passive and active isolation systems but were only able to effectively isolate the system from noise in the XY plane [11]. Liu et al. created an active noise cancellation methodology using a twin-tipped probe with the STM (see Figure 1.4a) [12]. One tip is used to measure the vibrations while the other records material measurements. An algorithm is used to actively cancel the vibrations detected in one tip from the noise in the other. This is a custom system though and does not provide a retroactive solution to vibration issues. Pabbi et al. created a similar active mitigation system that measures vibrations with a geophone placed further

up the STM and uses an algorithm to move the tip of the STM as a means of negating the incident vibration (see Figure 1.4b) [13]. However, this methodology operates best when only cancelling a single frequency.



(a) One of the tips records vibrations while the other collects data. The mechanical vibrations detected by one of the tips is processed and actively removed from data collected by the other. *Copied from ref* [12]



(b) Active noise cancellation that actively mitigates detected vibrations with an accelerometer and an ANITA processor. This system actively removes vibrations in real time but requires a custom processor to operate the tip. *Copied from ref* [13]

Figure 1.4: Active cancellation techniques interfere with the STM measurement itself to remove vibrations

Active cancellation techniques like these, impact the measurements taken by the STM directly. This makes troubleshooting difficult since there is not a control group to compare the method against. Bryce Primavera and Albert Chen originated and developed a post-processing methodology that keeps the original signal intact. By placing a geophone on the STM structure itself they simplified the propagation of vibrations to a single vertical axis. They characterized the relationship between vibrations detected by the geophone and the signal at the tip of the microscope as a vector transfer function that accounts for the attenuation and phase shift experienced by each frequency of vibration. Albert

created a calibration process that calibrated this transfer function before a material measurement. He then coupled it with the geophone signal recorded during the measurement to estimate the impact that vibrations had on the STM tip. This estimation was then subtracted from the measured tip data after the measurement was over. Albert was able to achieve 81.5% attenuation of root-mean-squared (RMS) value in noisy topographic measurements within a 0-300 Hz bandwidth [7] [8].

1.4 Targeted Users

Having worked directly with researchers in Hoffman lab, the users of a noise cancellation algorithm would be the researchers that collect and analyze topographic and spectroscopic data. Increasing the SNR of their data would allow them to better quantify an understanding of the quantum materials they research. The solution would be useful to all users of STMs, however since STMs vary in size and therefore isolation capabilities, those with noisier experimental conditions would see more benefit. Such is the case of tabletop STMs, these are available from commercial vendors like AFM Workshop and Nanosurf, but due to the lack of significant passive isolation systems incorporated they are very noisy [5] [6]. Many labs use the custom-built approach, where many passive isolation systems are in place to collect the best data possible. A problem that cryogenic STM users face is the amount of time they have for experimentation. The total time is typically restricted to 2-7 days, depending on the capacity of the liquid helium Dewar. The noise cancellation

solution means that the data can be collected faster, unlocking larger and more useful data sets.

1.5 Design Specifications

To evaluate whether or not this project was successful, a set of design goals were set in place based off of the problem discussed in Section 1.2. These design goals are detailed in Table 1.1 below. The first requirement is the implementation of both the vector cancellation and matrix cancellation schemes to remove noise from topographic and spectroscopic data. Both methods were implemented to ascertain which method is more effective, with the vector cancellation scheme outperforming the matrix cancellation scheme in all applications.

Noise is the result of uncontrolled changes in the data that deviate the recorded signal from its ideal value. In the application of cancellation schemes to topological data, I use a zero-centered root-mean-squared (RMS) value to quantify the difference between the measured data and data processed using both cancellation schemes. Previous ES100 student, Albert Chen, demonstrated >80% cancellation of topographic noise under noisy testing conditions. In this project I aimed to achieve that level of cancellation with the matrix cancellation scheme in both topographic and spectroscopic applications.

The bandwidth of the cancellation schemes must cover the bandwidth of recorded noise. The feedback system that records vibrations impacting the tip of

the STM has a bandwidth of 0-300 Hz, therefore a cancellation scheme must be able to cancel vibrations within this range. This criterion is met when I utilize the vector cancellation scheme, but the matrix cancellation scheme has a bandwidth of 50-400 Hz due to method of calibration I implement, detailed in Section 3.1.

A successful cancellation scheme needs to be usable by researchers. Popular interface languages are LabVIEW and a Python back-end; however I only wrote usable programs in Python. The data that is collected must have a large enough sampling rate that frequency characteristic can be discerned. In topographic noise cancellation, this means having a sampling rate above two times the bandwidth of the system at 600 Hz. In spectroscopic noise cancellation, this means having a sampling rate twice above the frequency by which the voltage bias is modulated, which can range from 1 kHz – 1.25 kHz.

Calibration for either scheme cannot negatively impact research in the lab. The STM is able to remain at proper testing temperatures for Hoffman lab between 2-7 days so the calibration time of the scheme must not exceed 3 hours.

Table 1.1: Technical Specifications

Requirements	Design Specification	Reasoning
Implementation	Cancellation in topographic and spectroscopic data	Researchers utilizes both measurements
Noise Level Attenuation	> 80% RMS value reduction in noise	Increase the SNR enough to enable faster data collection time
Noise Frequency Attenuation	0-300 Hz	Vibrations are typically between 0 and 120 Hz with higher harmonics
Usability	Executable in LabView/Python	Current researcher workflow mainly occurs within LabView and Python
Sampling Rate	> 2.5 kHz	Nyquist frequency must be over the modulation frequency (1 kHz – 1.25 kHz) during spectroscopic testing
Calibration Duration	< 3 hours	Helium maintains required temperature for 2–7 days

Chapter 2

System Design

Summary

To follow the specifications set in Section 1.5, the solution considers the theoretical and implementation levels. Section 2.1, Theoretical Background, addresses the scheme of the algorithm and the underlying math that dictates certain design choices. Section 2.2, System Implementation, addresses how the algorithm and measurement scheme interacts with the system. The build from these design choices is tested and analyzed later in the report.

2.1 Theoretical Background

2.1.1 Vector Cancellation

Since the geophone is not at the same position as the STM tip there is going to be some change in the vibration signature from what is recorded by the geophone and what is detected at the tip. These changes are in the form of an attenuation and phase shift of vibrations. Since the geophone and STM tip are both oriented to be maximally sensitive to Z-axis vibrations I simplify the model as a one-dimensional system (i.e. this assumes that any vibration experienced by the tip had to travel through the geophone first) [5]. I then can create a one-

dimensional, vector, transfer function with calibration measurements from the geophone's Z-axis data, $G_{cal}(t)$, and the Z-axis tip position, $Z_{cal}(t)$. The calculation of this function can be seen in Equation 2.1, the magnitude and phase of each frequency is found by taking the Fourier transformation (*FT*) of each set of data. I characterize the relationship between the geophone and STM tip by calculating the complex ratio (encoding the attenuation and phase shift) at each frequency. I perform this calibration scheme while the STM is in steady state, locked in the XY plane, and held at a constant current by the system feedback. Calibrating the transfer function while the system is in steady state ensures that there are no signals unaccounted for (i.e. motion from moving the STM tip along the XY plane can cause a response in the Z direction of the tip that was the result of a vibration).

$$T(\omega) = \frac{FT[Z_{cal}(t)]}{FT[G_{cal}(t)]} \tag{2.1}$$

Once I have calibrated the vector transfer function, I can apply it to data recorded during measurements. By recording the vibrations detected by the geophone, $G_m(t)$, simultaneously with the tip position, $Z_m(t)$, during a scan I am able to estimate the amount of noise as a result of those vibrations with the vector transfer function. The estimation of vibrational noise that comes from $G_m(t)$ can be seen in Equation 2.2. I then subtract this estimated $Z_{noise}(t)$ from the recorded tip position data to remove coherent vibrations from the signal, denoted in Equation 2.3.

$$Z_{noise}(t) = FT^{-1}[T(\omega) * FT[G_m(t)]] \quad (2.2)$$

$$Z_{signal}(t) = Z_m(t) - Z_{noise}(t) \quad (2.3)$$

2.1.2 Matrix Cancellation

A key assumption I made while calculating the vector transfer function is that each frequency of vibration has a linear attenuation and phase shift as it propagates from the geophone to the STM tip. However, there are propagation characteristics that are not considered in the vector transfer function. Such is the case of harmonic distortion, which can occur as the result of high amplitude vibrations [14]. Harmonic distortion is characterized as the appearance of a signal in the output of a system that was not present in the input signal. In the case of vibrations, a high amplitude vibration incident on a complex system, like the STM, can cause excitations at integer multiples of the vibration's frequency. The frequency of vibration in the input signal that causes these harmonic signals will be referred to as the 'fundamental frequency' in the rest of this paper. The key point here is that the magnitude of harmonic signals appearing in the output are the direct result of the magnitude of the fundamental frequency and are not the result of the magnitude of frequencies detected in the input signal. For example, a fundamental frequency of ω would be detected at the input and results in harmonic responses at 2*ω and 3*ω at some magnitudes. The magnitude of the signals at 2*ω and 3*ω are the result of the magnitude of the

fundamental frequency and not the linear response of those frequencies in the input signal. The vector transfer function does not make this distinction between harmonic and linear response. By making the assumption of a linear system the vector transfer function characterizes the magnitudes of the 2*ω and 3*ω signals as the linear response of those frequencies in the input signal. This becomes an issue if the magnitude of the fundamental frequency changes from what the vector transfer function was calibrated at since in reality the magnitude of the harmonic frequencies is directly related to the magnitude of the fundamental. If the magnitude of the fundamental frequency were to change, the vector transfer function does not take into account the resulting change in harmonic magnitude. However, a matrix transfer function can quantify this relationship between fundamental and harmonic frequencies. This type of transfer function assesses the response of not just the fundamental frequencies detected by the geophone but also the harmonic responses of the system as well. Each harmonic can be expressed as a coefficient $C_n(\omega)$, where the nth coefficient represents the n^{th} harmonic response of an incident frequency and $C_1(\omega)$ denotes the fundamental. The harmonic responses of each frequency can then be summed with the fundamental response to achieve a clearer picture of vibrations impacting the STM tip in high amplitude circumstances. The code that implements Equation 2.4 can be found in Appendix D.1.

$$Z_{noise}(\omega) = C_1(\omega)G_m(\omega) + C_2\left(\frac{\omega}{2}\right)G_m\left(\frac{\omega}{2}\right) + C_3\left(\frac{\omega}{3}\right)G_m\left(\frac{\omega}{3}\right) + \cdots \quad (2.4)$$

This matrix transfer function is calibrated differently than the vector since it would be difficult to differentiate between a harmonic response and a linear

response while the system is in steady state. To characterize the harmonic response, the STM system is excited at a known frequency, ω, and the system's response to that frequency is calculated. The system's response to the fundamental frequencies is found by driving the system with a frequency chirp over the desired frequency bandwidth. To isolate the response of the system due to the driven frequency from the surrounding signal I wrote code to reproduce a digital lock-in amplifier. This can be used to isolate the magnitude and phase of a signal in the recorded data relative to a reference signal. By using the known driving frequencies of the frequency chirp as a reference, the corresponding signal in both the STM tip and geophone can be calculated using Equation 2.5 (either signal represented by $S(t)$). A complex representation of the signal with respect to the reference frequency is found by multiplying the data by the reference signal, putting the product through a low-pass filter (LPF) with a cutoff frequency much less than ω, and then integrating over the time by which that reference frequency is driving the system [15].

$$LI_S(\omega) = \int_{t'}^{t'+T} LPF\left[e^{-it\omega} * S(t)\right] \tag{2.5}$$

I use Equation 2.5 to calibrate each coefficient at the corresponding frequencies in the chirp, shown in Equations 2.6-2.7. The method by which the system is driven is important as well. Ideally a pure tone would provide the driving force but in reality, the driving mechanism itself can produce a number of harmonics. To account for these harmonics in the driving signal, I use the fundamental response, $C_1(\omega)$, in Equation 2.7 to calculate the direct response in

the tip due to any detected signal at the harmonics of the driving signal. I then use each successive coefficient to remove the impact that harmonic responses to the input signal may have, i.e. while calculating the third harmonic response to a frequency ω I signals the signals generated by the fundamental response of the signal at *3*ω* as well as the second harmonic response at *1.5*ω* to ensure that the calculated harmonic distortion is the sole result of the signal at *ω*. Once I calibrate the coefficients and use Equation 2.4 to find $Z_{noise}(\omega)$, I subtract the estimated noise from $Z_m(t)$ just as with the vector cancellation scheme, shown in Equation 2.8. The code that implements Equations 2.5-2.7 can be found in Appendix D.2, and the code that implements Equation 2.8 can be found in Appendix D.1.

$$C_1(\omega) = \frac{LI_{Z_{cal}}(\omega)}{LI_{G_{cal}}(\omega)} \tag{2.6}$$

$$C_n(\omega) = \frac{1}{LI_{G_{cal}}(\omega)}\left[LI_{Z_{cal}}(n\omega) - \sum_{\substack{k\in\{\,integer\\ divisors\ of\ n\}\\ such\ that\ k<n}} C_k\left(\frac{n}{k}\omega\right) * LI_{G_{cal}}\left(\frac{n}{k}\omega\right)\right] \tag{2.7}$$

$$Z_{signal}(t) = Z_m(t) - \ FT^{-1}[Z_{noise}(\omega)] \tag{2.8}$$

2.1.3 Spectroscopic Cancellation

Spectroscopic data is collected under different testing condition than topographic data. These differences need to be accounted for a successful cancellation scheme to be made. One such difference is the use of a lockin amplifier to isolate the *dI/dV* signal from the surrounding data. While the tip position is locked in place the lockin amplifier modulates the bias *V(t)* at a high frequency, ω_{mod}, by a few millivolts and measures the resulting change in current, *I(V(t)),* giving the lockin signal, *LIY,* as an output from Equation 2.9. The bias is modulated according to Equation 2.10 [7]. This modulation of the voltage bias complicates the noise cancellation scheme.

$$LIY = \frac{1}{T}\int_{t'}^{t'+T} I\big(V(t)\big)\cos(\omega_{mod}t)\,dt \tag{2.9}$$

$$V(t) = V_{DC} + V_{AC}\cos\left(\omega_{mod}t\right) \tag{2.10}$$

As a result of modulating the voltage bias following Equation 2.10 an aliasing affect in the current, *I(V(t)), and LIY* signals appears. This effect exhibits itself as a repeated noise signature of ω_{noise} *at* $\omega_{mod} \pm \omega_{noise}$. Since this shifted noise term is the result the voltage modulation, I use Equation 2.11 to calculate an aliasing transfer function, which can be used to estimate the amount of noise that is the result of the aliasing effect using Equations 2.12. The implementation of Equations 2.11-2.12 into code can be found in Appendix D.5.

$$A(\omega) = \frac{I(|\omega - \omega_{mod}|)}{I(\omega)} \quad |\omega \leq 2 * \omega_{mod} \tag{2.11}$$

$$I_{alias}(\omega) = A(|\omega_{mod} - \omega|) * I_m(|\omega_{mod} - \omega|) \tag{2.12}$$

$$\begin{cases} RE(I_{alias}(\omega)) = RE(I_{alias}(-\omega)) \\ IM\big(I_{alias}(\omega)\big) = -IM(I_{alias}(-\omega)) \end{cases}$$

To cancel noise from current data I take advantage of the relationship between current and tip position and exponent laws using what I will refer to as 'Method 1'. This method involves first calibrating the matrix transfer function from *G-Z* and then mathematically calculating its transfer to current, *I*. The current signal is exponentially related to the Z position following Equation 2.13. By dividing the recorded current value by the exponentiated negative product of the estimated Z_{noise} and the kappa value, I can effectively remove the noise term from the measured current. This is shown in Equation 2.14 and results in the cancelled current value $I_{sig}(t)$. The kappa value, κ, is found experimentally by moving the tip of the STM away from the sample along the Z axis, shown in Figure 2.1. The current that is recorded during that motion is logarithmically related to Z and is used to determine κ. The code that implements Equations 2.13-2.14 can be found in Appendix D.3.

$$I\big(V(t)\big) = I_0(V(t))e^{-\kappa Z_{noise}} \tag{2.13}$$

$$I_{sig}(t) = \frac{I(V(t))}{e^{-\kappa Z_{noise}}} = I_0\big(V(t)\big) \tag{2.14}$$

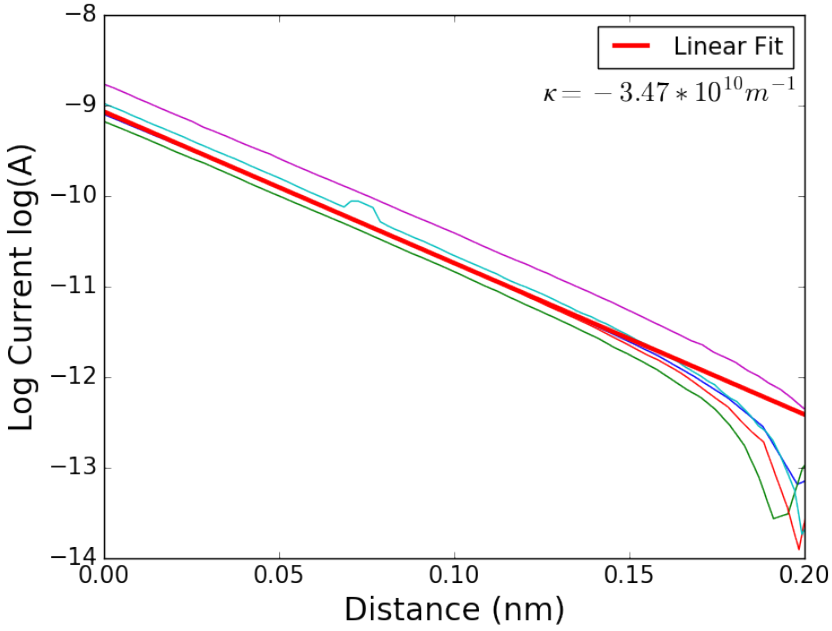


Figure 2.1: Kappa is experimentally calculated. By withdrawing the tip from the sample in the Z direction, the measured current gives a logarithmic relationship to Z, which is used to determine κ

Method 1 can be extended into cancelling noise from LIY data which I will refer to as 'Method 2'. Under the assumption that $\omega_{noise} << \omega_{mod,}$ I can consider that any noise produced by vibrations is at a constant level over the period of integration and can therefore be taken out of the integral following Equation 2.15. With the noise term outside the integral I am able to just divide the LIY_m data by the exponential noise term and remove it, shown in Equation 2.16. The code that implements Equation 2.15-2.16 can be found in Appendix D.6.

$$LIY_m = \frac{1}{T}\int_{t'}^{t'+T} I_0(V(t))e^{-\kappa Z_{noise}}\cos(\omega_{mod}t)\,dt \qquad (2.15)$$

$$\approx \frac{e^{-\kappa Z_{noise}(t)}}{T} \int_{t'}^{t'+T} I_0(V(t)) \cos(\omega_{mod} t)\, dt$$

$$LIY_{sig} = \frac{LIY_m}{e^{-\kappa Z_{noise}}} \approx \frac{1}{T} \int_{t'}^{t'+T} I_0(V(t)) \cos(\omega_{mod} t)\, dt \quad (2.16)$$

2.2 System Implementation

2.2.1 Overall System

Researchers initiate a scan with the STM using the Nanonis Scan Monitor, which then produces an image automatically from that measured data. An implementation of this project would ideally require it to be coupled with the Nanonis Scan Monitor as to not add extra steps to material measurements.

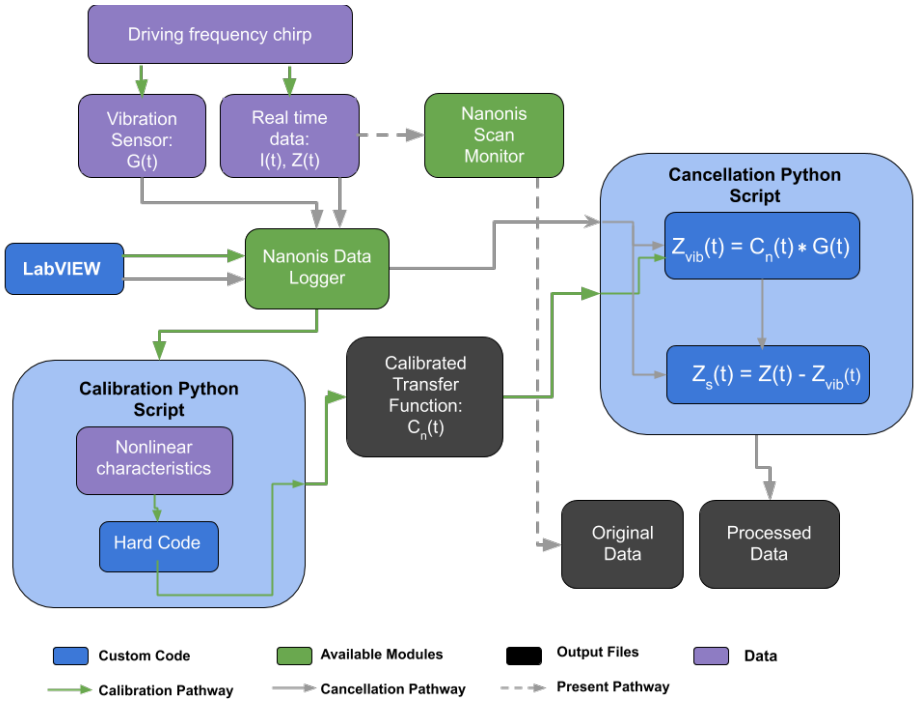


Figure 2.2. System architecture showing the steps data processing taking between Nanonis, LabVIEW, and Python. The green pathway shows the steps taken to calibrate a transfer function: the user activates the LabVIEW interface to instruct the Nanonis Data Logger to collect calibration which is then used by the Python code to generate a calibrated transfer function. The grey pathway shows the steps taken to cancel vibrational noise from measured data: once the user instructs that a scan should be taken the transfer function is then used to estimate the vibrations in the STM tip with the geophone data and subsequently removed to produce a clearer image.

2.2.2 Python Scripts

The calibrations python script operates in three steps: loading data collected during calibrations, processing the data, and creating a transfer function to be used later. The cancellation python script operates in four steps: loading the data collected during a scan, loading the calibrated transfer function, processing the data, returning the data without the estimated vibrational noise. The data is loaded using the respective file names logged by the Nanonis Data Logger. Once

loaded in 1D numerical arrays, the Z-position data is parsed through several functions that remove tip changes as well as piezo and thermal drifting seen in Figure 2.3. The transfer function is then calibrated using Equations 2.6 and 2.7 and saved as a matrix of complex frequency responses in a file that the cancellation script can access.

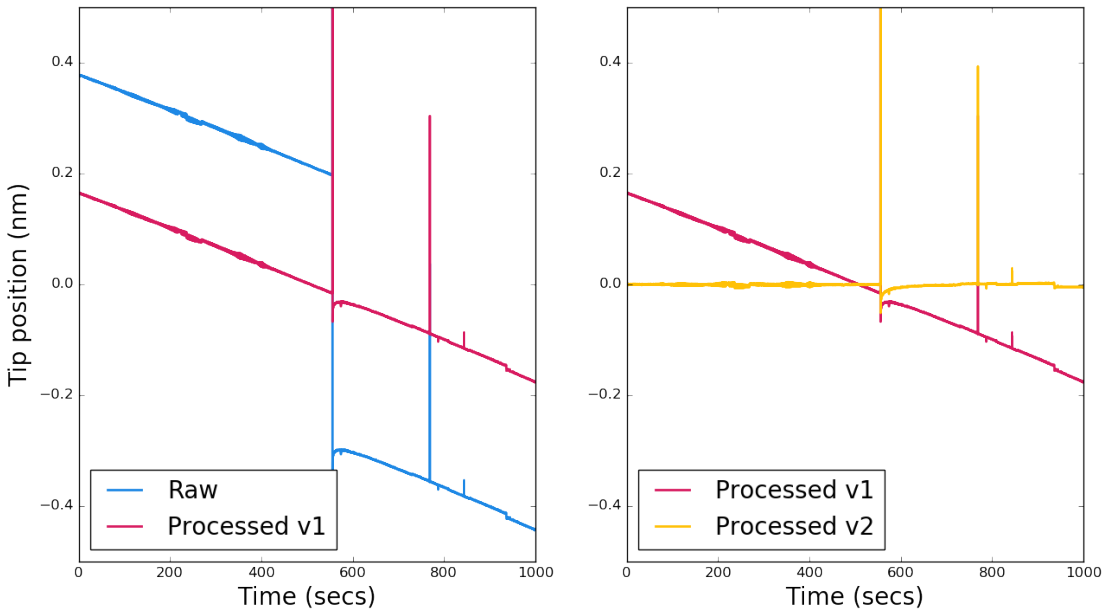


Figure 2.3. Detecting and eliminating tip change, and thermal drift. Tip changes are exhibited as steep shifts in the Z position of the tip that occur due to changes in tunneling location with the sample (Left). Thermal drifting exhibits itself as the sloping of the Z position data in the background, removed with a 2nd degree polynomial (Right). *Data shown was collected a 10 kHz sampling rate, driven with a 62 Hz sine wave from a speaker.*

The cancellation script operates similar to the calibration script steps: load scan data from the Nanonis Data Logger, process the data to remove detected vibrational noise, and save a new set of data with no noise. The script loads the calibrated matrix transfer function and scan data. The data is then processed,

assessing $Z_{noise}(t)$ from the measured geophone data and then removing that noise from $Z_{signal}(t)$, using equations 2.4 and 2.8.

Chapter 3

Topographic Noise Cancellation

Summary

I tested both noise cancellation schemes under noisy conditions to quantify its ability to reduce topographic vibrations that impact the position of the STM tip. Section 3.1 describes the calibration process of the matrix transfer function, detailing some shortcomings and how they are addressed. Section 3.2.1 describes the testing and analysis involved under noisy conditions with the result being the scheme achieved 52.6% ± 3.4% RMS value reduction. The algorithm is based on the assumption of coherent, high amplitude vibrations impacting the microscope. I did not apply either scheme under ambient operating conditions at the time of this project's completion.

3.1 Coefficient Calibration

To calibrate the matrix transfer function I drove the system with a stepwise frequency chirp delivered from a lockin amplifier at 150 mV through a speaker placed on the STM table seen in Figure 3.1. I then calculated the harmonic

response of the STM's tip position to each frequency in the chirp using coefficients as previously discussed in Section 2.1.2.

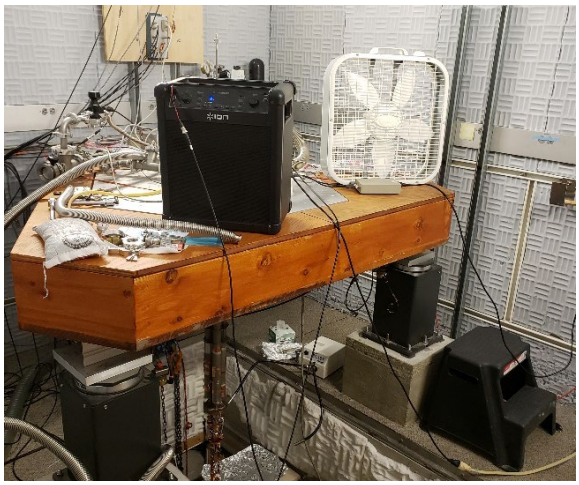


Figure 3.1. Calibrating the harmonic response of the STM with a speaker. The 8" speaker placed on the wooden table plays sine wave that sweeps in frequency from 50 - 400 Hz for calibrations. With a designed frequency step of 0.01 Hz and a step length of 0.1 seconds this calibration data set took ~1 hour to collect.

However, I did not collect the proper calibration data to accurately calculate the matrix coefficients. In order to achieve a desired frequency resolution in the coefficients of 0.01 Hz in a reasonable amount of time I set the step length of the frequency chirp to be 0.1 seconds. During analysis, after the data was collected and access to the STM was no longer available for testing, I discovered that this 0.1 second step length does not allow Equation 2.5, the digital lockin amplifier calculation, to achieve 0.01 Hz resolution. In an attempt to remedy this issue, I stitch together 30 frequency steps to perform the lockin calculation on. This approximately resolves the frequency resolution issue but proposes its own set of issues. I calibrated the coefficients where only the center

of the stitched steps is equal to the reference frequency in the lockin calculation, therefore with every frequency step change away from the reference frequency the signal becomes out of phase and improperly mixed with its reference signal. This results in a coefficient calculation that does not accurately reflect the attenuation and phase shift of the data.

Figure 3.2 shows the format that a calibrated matrix transfer function takes. This figure is useful in understanding what the coefficients are achieving. Looking at the x-axis, the geophone has an input signal at some frequency ω, this results in a tip response at frequency ω but by following the vertical column there is also a clear response in the tip at 2ω, 3ω, 4ω, and 5ω.

Figure 3.3 demonstrates a more exact picture of what the coefficients look like. Each coefficient covers less of the of the frequency bandwidth since they are harmonics of the frequencies (e.g. $C_2(\omega)$ appears as only spanning 50-200 Hz when in fact it estimates the response the STM tip will have from 100-400 Hz.

Figure 3.4 compares the vector transfer function and the first coefficient of the matrix transfer function. I calibrate the vector using calibration data collected while the system was being driven by a 62 Hz sine wave for 300 seconds at the same amplitude as the frequency chirp that was used to calibrate the matrix transfer function. This comparison shows that there is something wrong with the matrix transfer function as it is consistently an order of magnitude below the transfer function over most of the frequency bandwidth. The only time in which the two traces come close is at 62 Hz, but matrix is still a factor of 2 below the vector transfer function.

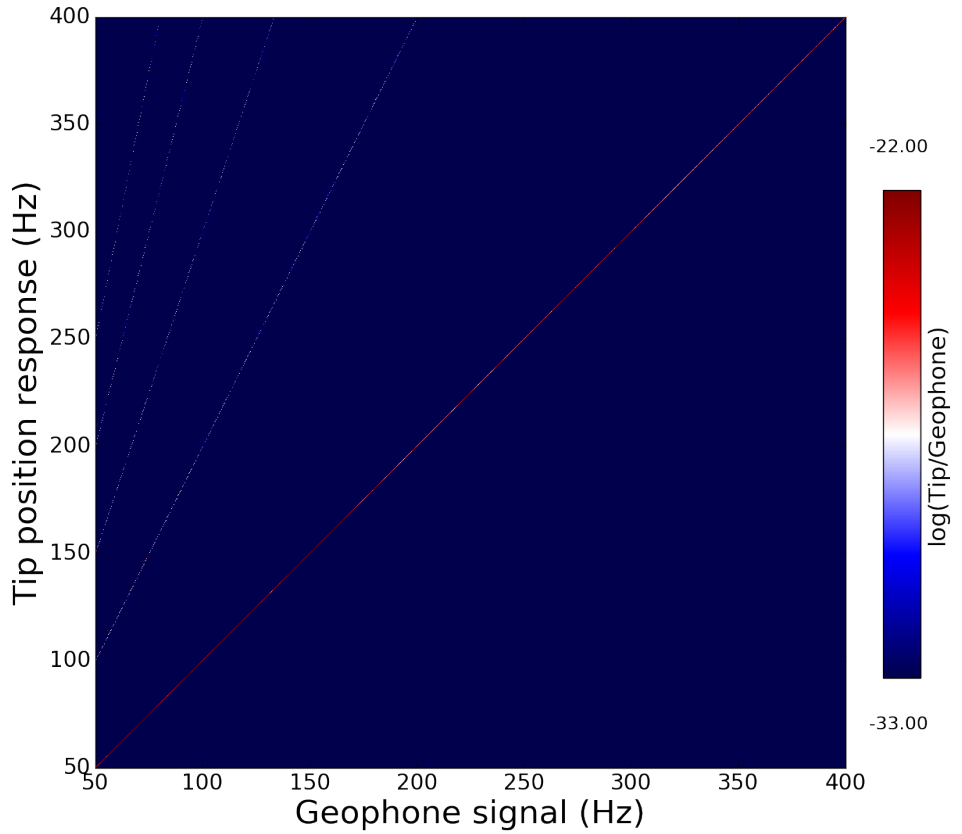


Figure 3.2: Calibrated coefficient matrix. The diagonal coefficients show that different frequencies will respond differently to an incident signal from the Geophone. *This matrix was calibrated using data that was collected at 20 kHz, down sampled by a factor of 2. Each pixel on the diagonal lines are the result of calculating the harmonic response of the STM tip position due to an input signal from the Geophone. The frequency chirp from 50-400 Hz drove the speaker at 150 mV with a frequency step of 0.01 Hz and length of 0.1 secs. 30 frequency steps were stitched together to achieve an appropriate frequency resolution.*

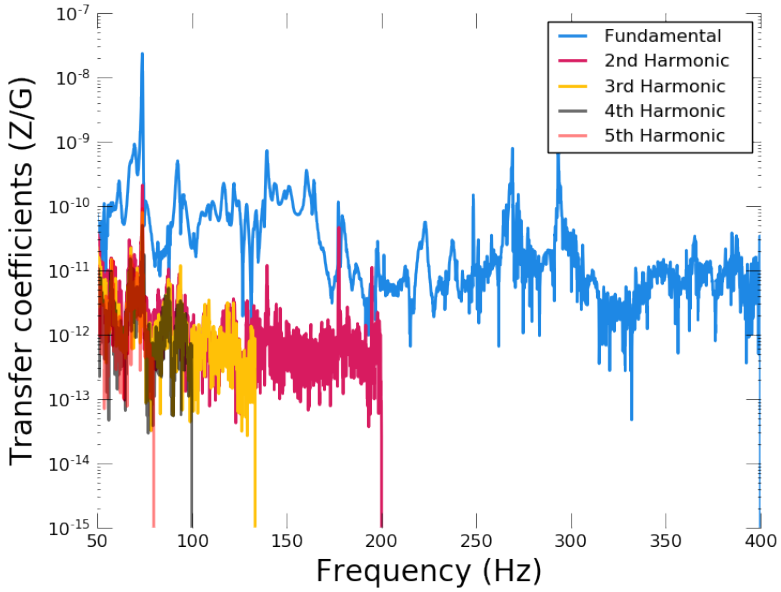


Figure 3.3: Calibrated coefficients in frequency domain. The coefficients, although related, have different shapes due to the differences in the harmonic responses. *The data that was used to this figure is the same that was used in making Figure 3.2*

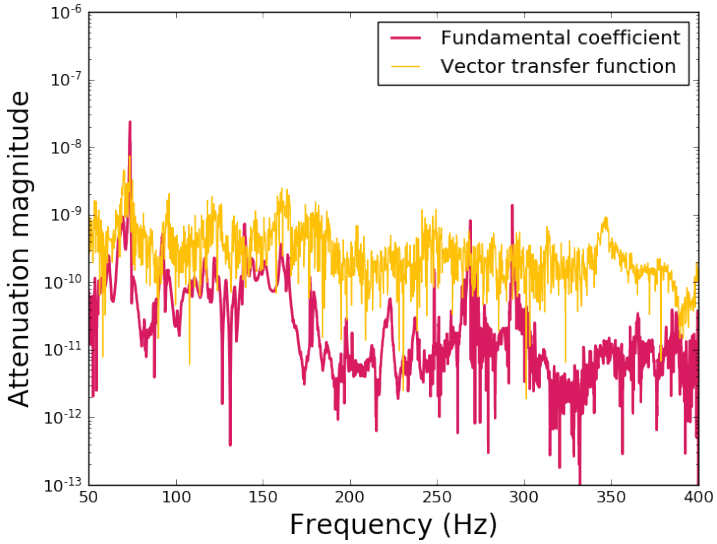


Figure 3.4: Comparison of the fundamental coefficient and vector transfer function. The comparison of these two transfer functions indicates that something is wrong with the fundamental coefficient. This is supported by the poor performance of the matrix transfer function in Section 3.2. *The data that was used to create the fundamental coefficient is the same as Figures 3.2 and 3.3. The data that was used to create the vector transfer function was collected at 20*

kHz and down sampled to 10 kHz. The system was excited by a 62 Hz sine wave while the calibration data was being recorded. The units are on the scale of ~1e-9 due to the units of the geophone and tip position, which are mV and pm respectively.

3.2 Effective Noise Reduction

3.2.1 Driven Noise Reduction

Using the matrix transfer function, I am able to reduce the RMS value of noisy data, 62 Hz sine wave driven from the speaker, by 52.6% ± 3.4%. This metric was calculated over 80 seconds of test data by finding the RMS value of the raw data and the cancelled data every 5 seconds, then calculating the mean and standard deviation between those calculated values. Using the vector cancellation scheme though, I am able to reduce the RMS value by 85.9% ± 10.7%, found using the same process as the matrix reduction calculation. This large of a value is likely due to the fact that the driven test data has very large coherent vibrations. The time trace of the matrix cancellation and vector cancellation can be seen in Figure 3.5 and shows that the vector scheme outperforms the matrix scheme by a large degree in the tested circumstances.

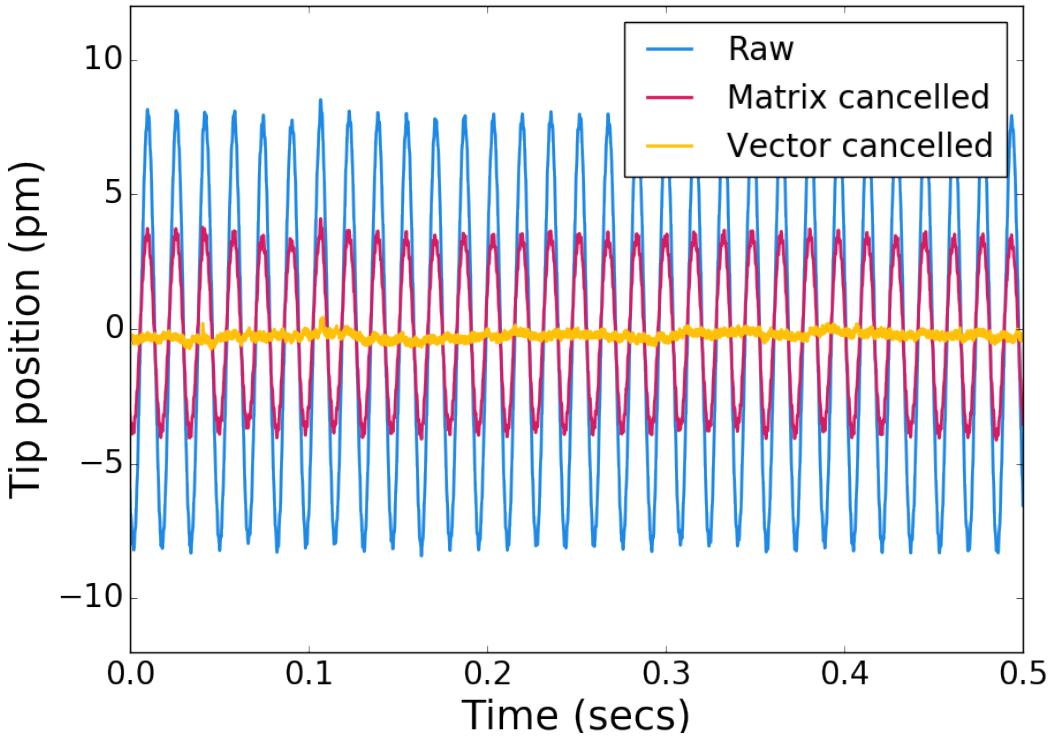


Figure 3.5. A 52.6% RMS value reduction in noise between the measured and matrix processed Z trace. The processed signal (red) is seen to have slightly suppressed the most prominent driven vibration while also staying in phase with the raw signal (blue). The vector processed signal (yellow) reduced the driven vibrations to a much greater degree than the matrix, achieving a 85.9% RMS reduction during the time period shown. *Data shown was collected a 10 kHz sampling rate, driven with a 62 Hz sine wave from a speaker.*

Shown in Figure 3.6, the Fourier transform of the 3 signals show that the vector processed data has reduced all prominent peaks in the frequency domain while the matrix processed data only successfully reduces the fundamental peak slightly and incorrectly adds noise to the other prominent frequency peaks. This incorrect attenuation and phase shift estimation is likely the result of how I calibrated the coefficients. Figure 3.7 demonstrates the miscalculation of phase and amplitude in the matrix noise estimation. The noise signal that I generate

with the matrix transfer function is ~20 times smaller in amplitude and ~45° out of phase with the unprocessed data. The signal I generate with the vector transfer function does not have this issue and almost exactly aligns itself with the unprocessed data.

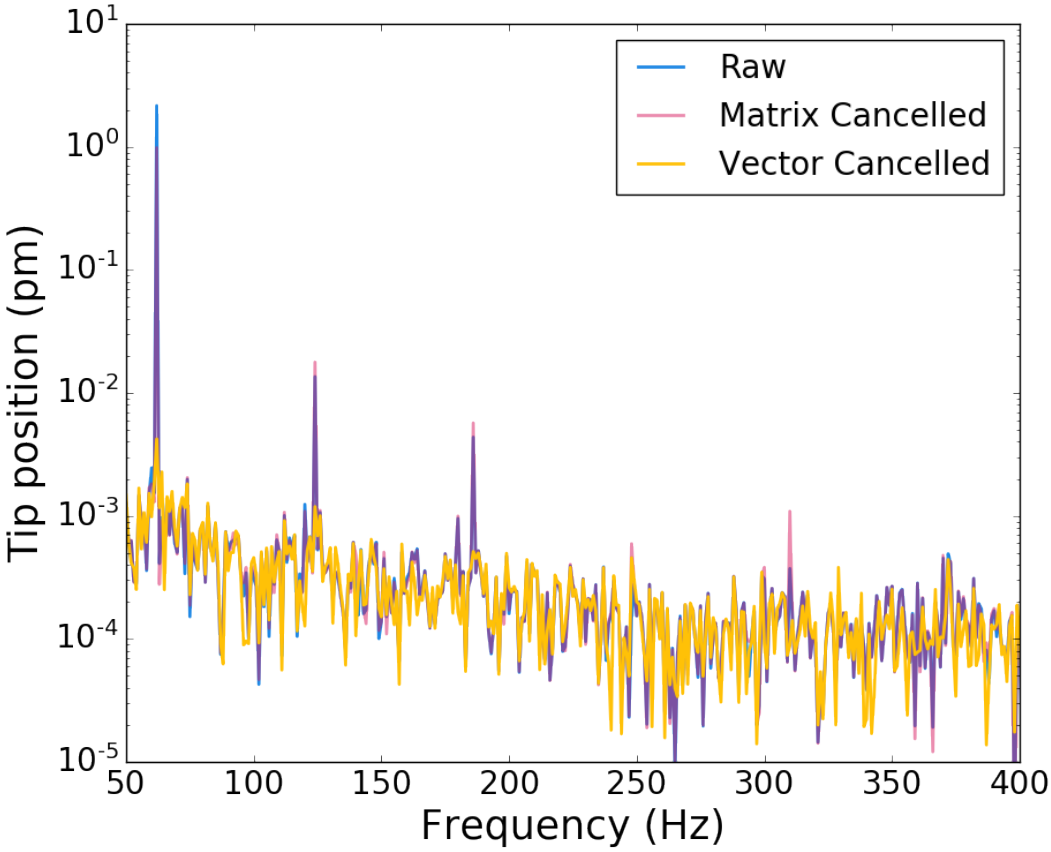


Figure 3.6: The matrix transfer function adds harmonics to the signal instead of removing them. While the matrix transfer function reduces the fundamental peak (The large peak seen on the left at 62 Hz) slightly but harmonic peaks to the signal are added to the signal. This is likely due to both a misestimation of magnitude during coefficient calibration and a miscalibration of phase shift demonstrated in Figure 3.7. The vector cancellation is able to reduce the fundamental peak by nearly 3 order of magnitude and reduce the other large peaks seen in the figure. *Data shown was collected at a 10 kHz sampling rate, averaging together the Fourier transform of every 2 seconds of data for 80 seconds to achieve 0.5 Hz resolution.*

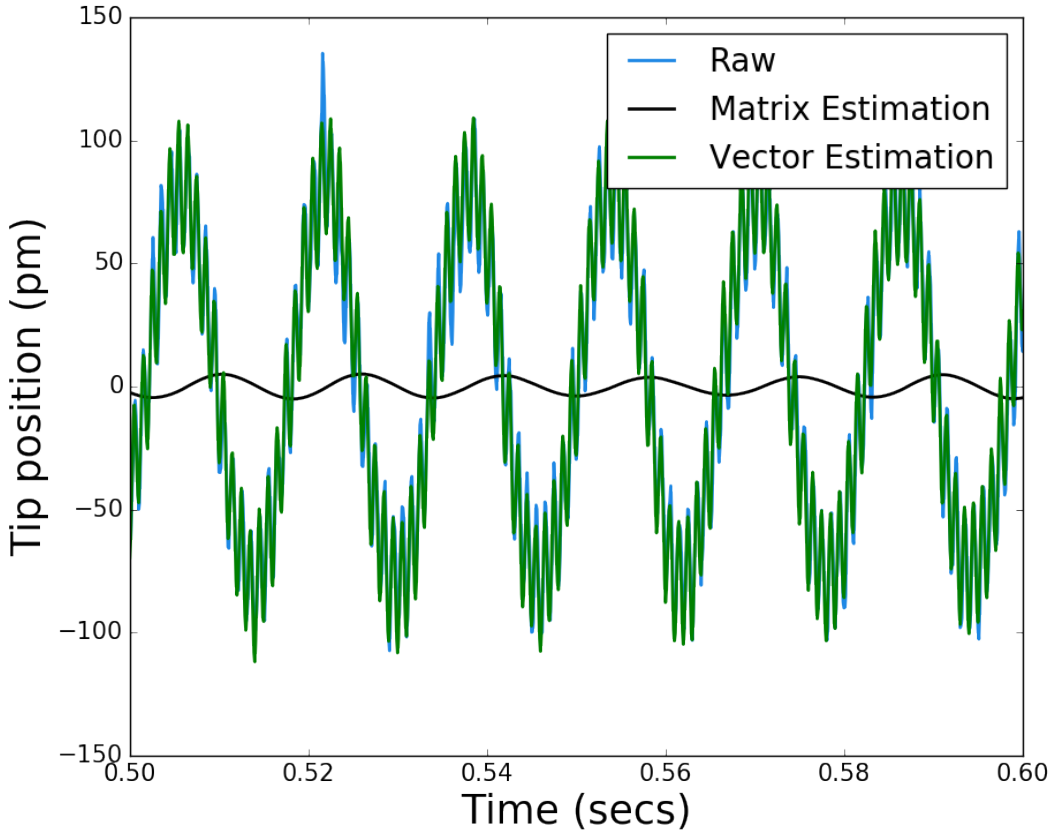


Figure 3.7: The magnitude and phase of noise in the raw data is not accurately estimated. The noise estimated by the vector transfer function is almost direct on top of the raw data, indicating a great majority of the signal is noise. However the noise estimated by the matrix transfer function is ~45° out of phase with the raw data and underestimating it by a factor of ~20. The result of being out of phase is that some signals that the are supposed to be cancelled end up being amplified, see in 124 Hz and 186 Hz peaks of Figure 2.6. *Data shown was collected at a 10 kHz sampling rate, averaging together the Fourier transform of every 2 seconds of data for 80 seconds to achieve 0.5 Hz resolution*

Chapter 4

Spectroscopic Noise Cancellation

Summary

Spectroscopic data provides details about the electronic structure of tested materials. The material's electronic density of states (DOS) is found by sweeping the voltage bias applied to the sample while also modulating that bias by a few millivolts. A lockin amplifier is then used to measure the corresponding change in current. Vibrational noise impacts this measurement particularly when the frequency of vibration is very low and very close to the modulation frequency. Section 4.2.1 demonstrates the application of the noise cancellation algorithm to the spectroscopic data, both the current data *I(t)* as the voltage is swept and the corresponding lockin measurement *LIY*. The result of this methodology is a 12.2% reduction in LIY data of the standard deviation from all the voltage sweeps averaged together.

4.1 Effective Noise Reduction

4.1.1 Driven Noise Reduction

To verify that the scheme can reduce vibrations in spectroscopic data, I drove high amplitude vibrations into the system with the lockin amplifier on while recording current and LIY data, as well as the geophone and tip position signals. Using the setup of the speaker on top of the STM (see Figure 3.1) to drive a 62 Hz sine wave into the system I determined the performance of the cancellation scheme when applied to different spectroscopic measurements. The voltage was swept from -500 mV to 500 mV at 10 mV voltage steps. Each of the 500 sweeps performed took ~1.2 seconds 500 times, the resulting current from a voltage sweep is shown in Figure 4.1. I used the averaged current data to center each

sweep about 0 and calculate the RMS value reduction for each scheme when using Method 1 over 499 voltage sweeps. I was able to achieve 32.6% ± 1.3% RMS value reduction with the matrix transfer function and 41.8% ± 1.6% RMS value reduction with the vector transfer function.

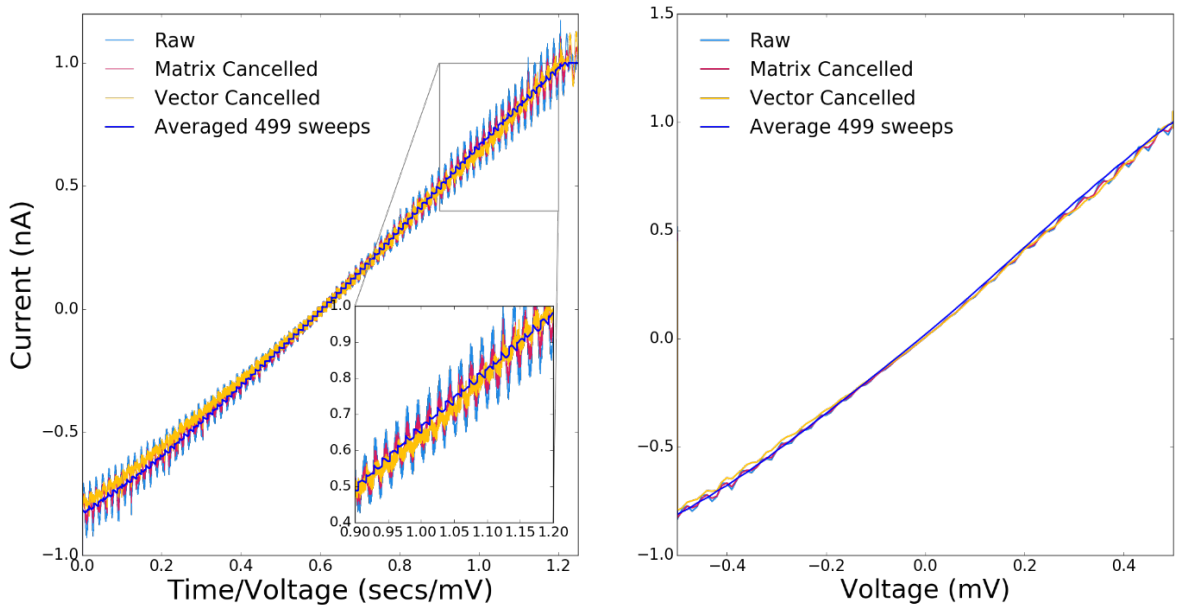


Figure 4.1: RMS value of current during a voltage sweep reduced by 32.6% and the corresponding I/V curve using Equation 2.19 (Method 1). Notice the larger current amplitude at the beginning and end of each sweep (noticeably on the left but also on the right plot). This occurs as the current value scales with the supplied voltage bias, following Equation 2.11. *Data shown was collected at a 20 kHz sampling rate, down sampled by factor of 2. Data was collected with the voltage bias also being modulated by 10 mV at 1 kHz and the system was being driven by a 62 Hz sine wave.*

Using Method 2 I cancelled noise in LIY_m data with both cancellation schemes. The RMS value of the LIY data is not a viable metric to use in assessing the effectiveness of either scheme. This is because the value is not constant with voltage as current and tip position are with time. So to determine the effectiveness of the cancellation schemes I calculated the reduction in the

standard deviation from the average LIY data over every sweep collected. Figure 4.3 show that cancellation schemes were able to bring the raw LIY data closer to the averaged sweep. By calculating the average standard deviation from the average sweep for each set of data over every voltage sweep I calculated the amount the standard deviation decreased due to being processed by each cancellation scheme. Using the matrix transfer function I was able to reduce the standard deviation by 12.2% ± 9.3% based on the voltage bias. Using the vector transfer function I was able reduce the standard deviation by 41.2% ± 0.8%. Looking at the Fourier transform in Figure 4.4 the vector cancellation scheme clearly performed better than the matrix transfer function in reducing the magnitude of the fundamental peak at 62 Hz.

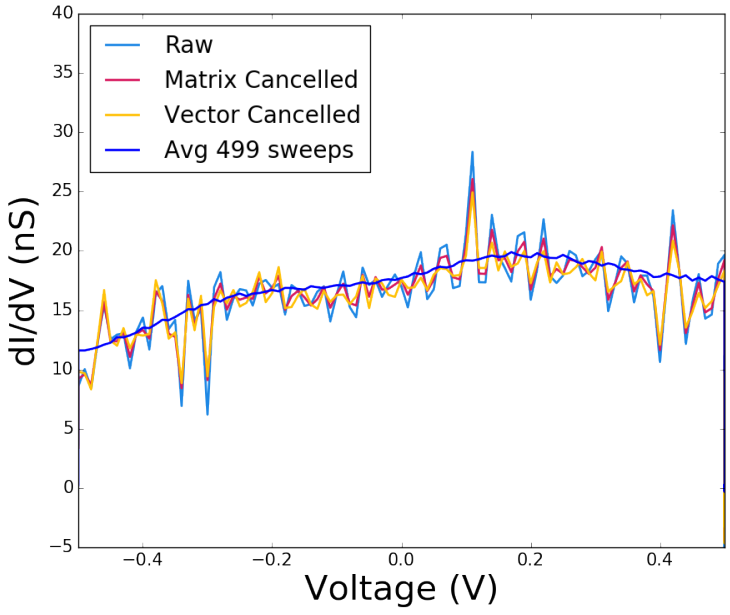


Figure 4.3: Reduction in the standard deviation from the average swept value indicates reduction in vibrational noise. Both sets of processed data show a reduction in the amplitude of peaks toward the averaged trace. *Data shown was collected at a 20 kHz sampling rate, down sampled by a factor of 2, and integrated*

over the period of modulation (1/1000 sec). Data was collected with the voltage bias also being modulated by 10 mV at 1 kHz and the system was being driven by a 62 Hz sine wave.

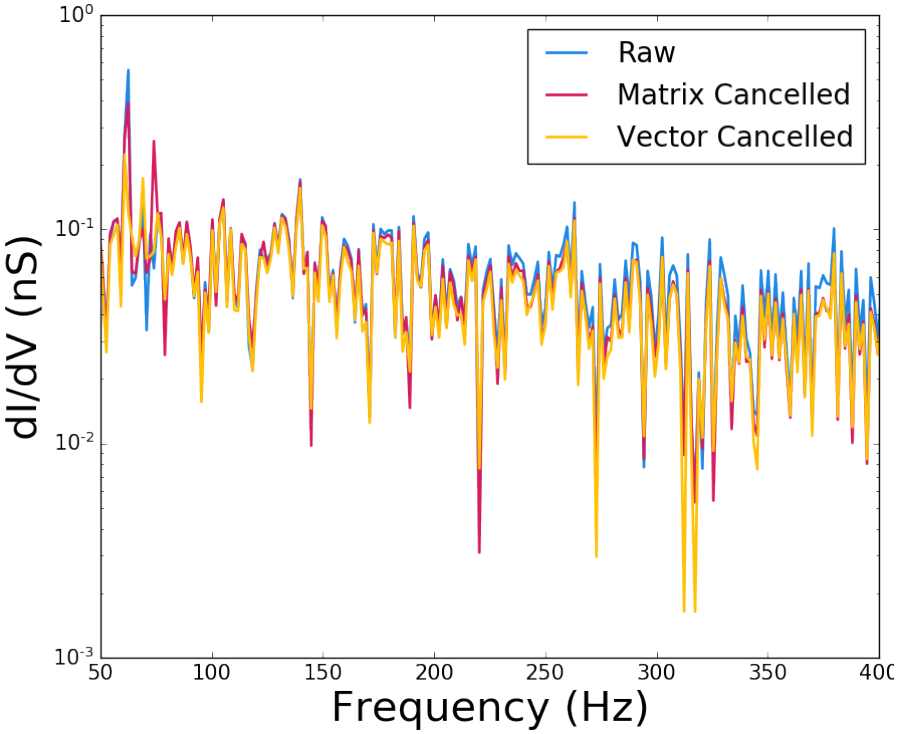


Figure 4.4: Driving frequency reduced in LIY data. The principle frequency (62 Hz) seen in the Fourier transform of LIY data shows that the principle frequency has been reduced. *Data shown was collected at a 10 kHz sampling rate, down sampled by a factor of 2 and integrated over the period of modulation (1/1000 seconds). Data was collected with the voltage bias also being modulated by 10 mV at 1 kHz and the system being driven by a 62 Hz sine wave. The figure displayed shows the Fourier transform of LIY data averaged together over 499 voltage sweeps*

4.1.2 Spectroscopic Aliasing Reduction

Discussed in Section 2.1.3, spectroscopic aliasing arises due to the modulation of the voltage bias supplied to the sample being scanned. This modulation shifts low frequency noise about the modulation frequency so that vibrational noise is seen at ω_{noise} and $\omega_{mod} \pm \omega_{noise}$. Using Equations 2.11 and

2.12, the relationship between shifted and low frequency noise is found and removed from measured current data (see Figure 4.4).

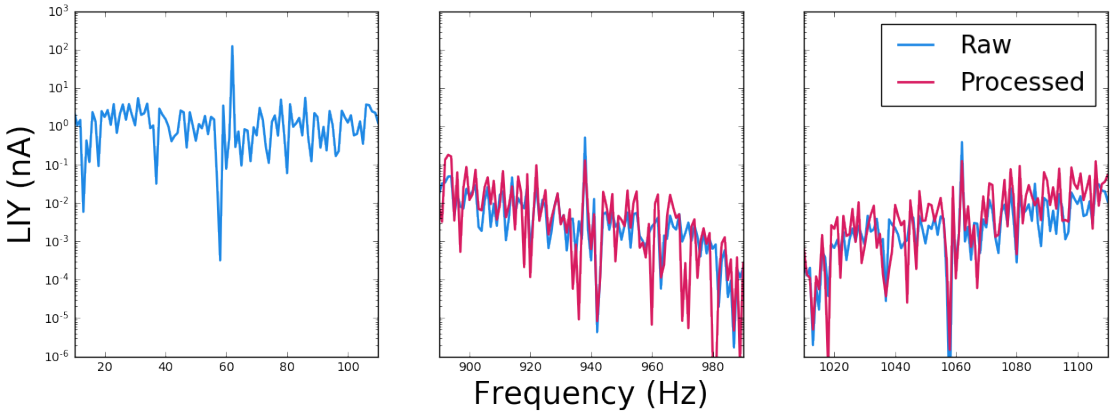


Figure 4.5: Low frequency vibrations are shifted about the modulation frequency produced by the lockin amplifier. The lockin amplifier shifts the low frequency vibration signatures 62 Hz (left) about the frequency it modulates the voltage bias of the sample, by 10 mV at 1 kHz. This can be seen in peaks at 948 Hz (middle) and 1062 Hz (right). This nonlinear phenomenon is accounted for with an additional transfer function. *Data shown was collected at a 20 kHz sampling rate, down sampled by a factor of 2. The Fourier transforms of every 2 seconds of data for 80 seconds were averaged together to achieve 0.5 Hz resolution but then was down sampled to show 1 Hz resolution for less noise.*

Chapter 5

Conclusions

The STM is an atomic-resolution microscope that can detail the atomic and electronic structure of conductive materials. These microscopes are

incredibly sensitive to incident vibrations from their surrounding environment due to the scale of material characteristics they define. I used the post-processing method described in this thesis to quantify the response of the STM, in the Hoffman lab, to vibrations I drove with a speaker. By recording vibrations with a geophone, the proposed schemes can estimate the impact vibrations have on the STM tip and then remove them after a measurement has been completed. The post-processing method strengthens the signal recorded by the STM, allowing the potential for faster scan times. Future work includes optimizing the matrix transfer function and methodology to widen its frequency bandwidth and initialize a more consistent calibration scheme. It would also be noteworthy to determine how effective the matrix transfer function remains over time. This algorithm and measurement scheme have the potential to be applied to other sensitive instruments since it is applied retroactively along with other passive vibration mitigation systems and offers a robust vibration mitigation solution.

5.1 Project Success

I was not able to achieve the design specifications set forth by this thesis. I achieved 52.6% ± 3.4% reduction in RMS values of topographic data where the design goal was to achieve >80%. I did not successfully remove harmonics from

the test data, and instead added harmonic noise. I achieved 32.6% ± 1.3% RMS reduction in current data while the voltage was being swept where the design goal was to achieve >80%. I achieved 12.2% ± 9.3% standard deviation reduction while using the matrix cancellation scheme and 41.2% ± 0.8% standard deviation reduction while using the vector cancellation scheme. The noise attenuation was achieved over a 50–400 Hz bandwidth where the design goal was to be 0-300 Hz. The solution does not include a user interface through LabVIEW, but the Python back-end allows the potential to be combined with an existing interface framework. There is much room for future work.

5.2 Future Work

To improve the solution and meet the design specifications there are three main areas of work. The first addresses the lack of low frequency attenuation. In this project, a speaker was used to drive the STM at known frequencies to measure the system's response. These driving frequencies were limited by the ability of the speaker to play low frequencies. Using a linear actuator to drive the system at lower frequencies would allow the algorithm to estimate the system's response.

Second, optimization of the matrix transfer function would allow for more complete attenuation of vibrations. In this project, I collected one set of data that could be used for the matrix calculation, the shortcomings of this data set are discussed in Section 3.1. To optimize the calibration process, there is the potential to drive the system with a stepwise frequency chirp where the step time

is determined by the period of the driving frequency and not a set time limit. This would mean that lower frequencies would take more time to calibrate than higher frequencies, ultimately saving more time than if a set step length were used. Being able to dictate the number of periods driving the system at a given frequency would subsequently increase the accuracy of Equation 2.7.

The third area involve characterizing the effectiveness of the matrix transfer function over time. By collecting multiple calibration sets of data over a few days, the difference between the matrix coefficients could be quantified.

Appendix A

Spectroscopic data cancellation

A third method of spectroscopic cancellation would be that of directly calibrating a transfer function from *G-LIY* [7]. Unfortunately, I was not able to implement this method for the project due to a tip change in the data set that added a considerable amount of noise to the data. I therefore used the data set that was collected to calibrate the matrix transfer function but I had not modulated the voltage bias, thereby not producing *LIY* data. In applying this

scheme to spectroscopic data an approximation needs to be made. This approximation is that the magnitude of noise in the signal is constant during the period by which the lock-in is integrating. This allows the noise component of $e^{-\kappa Z}$ to be moved outside of the integral, such as in Method 2. This is valid whenever the frequency of vibrations is much less than the frequency of modulation, $\omega_{noise} \ll \omega_{mod}$. This is a valid approximation since most environmental vibrations are below 120 Hz and the lock-in amplifier modulates the bias by ≤ 1 kHz. In the vector transfer function, a second approximation is made that the effects of the exponential relationship between current and position are a linear relationship. In looking at the Taylor expansion for an exponential function, $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$, given that x is a periodic function, noisy environments (i.e. vibrations do not have a small amplitude, on the order of picometers, see Figure A.1*)* will exhibit harmonics in the current data regardless of if they manifest in the system itself following Equation A.2.

$$LIY_m \approx \frac{e^{-\kappa Z_{noise}}}{T} \int_{t'}^{t'+T} I_0\big(V(t)\big) \cos(\omega_{mod} t)\, dt \tag{A.1}$$

$$\sin(\omega)^n = \sin(n\omega) \tag{A.2}$$

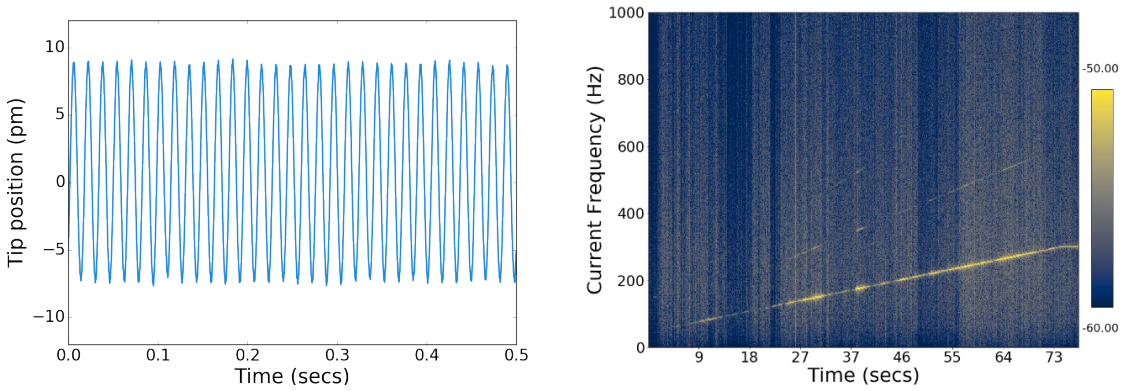


Figure A.1 High amplitude situations do not allow a linear approximation of the exponential relationship between tip position and current. The amplitude of tip position data is ~16 pm (left). This results in harmonic signals being detected during a frequency chirp in *I(t)* (right) shown as off diagonal lines of the most prominent signal in the spectrogram. *Data shown was collected at a 20 kHz sampling rate, down sampled by a factor of 2. The data on the right plot was driven by a stepwise frequency chirp from 50-400 Hz with a frequency step of 0.01 Hz every 0.1 secs. The data on the left was driven by a 62 Hz sine wave.*

Since the *LIY* signal changes along with the bias voltage so does the magnitude of the noise in the current. However, it is not feasible to calibrate a transfer function, matrix or vector, at each voltage for cancellation (sweeping a voltage from -500 mV – 500 mV with 10 mV steps would result in ~100 transfer functions). Instead, a scaling factor must be applied that is proportional to the DC current change. The new transfer function, Equation A.3, considers the LIY_{signal} coefficient in Equation A.4 by dividing the matrix coefficients by the average current value that the they were calibrated at. When LIY_{noise} is then estimated, the calculation is multiplied by the average current value during that step of the voltage sweep. This results in an I_m/I_{cal} scaling factor that corrects for the

magnitude change with voltage. The estimated LIY_{noise} can then be subtracted from the measured LIY_m data shown in Equation A.5.

$$C_n(\omega) = \frac{1}{LI_{G_{cal}}(\omega)} * \frac{1}{\overline{I_{cal}}} *$$

$$\left[LI_{LIY_{cal}}(n\omega) - \sum_{\substack{k \in \{ integer \\ divisors\ of\ n \} \\ such\ that\ k<n}} C_k\left(\frac{n}{k}\omega\right) * LI_{G_{cal}}\left(\frac{n}{k}\omega\right) \right] \tag{A.3}$$

$$LIY_m \approx LIY_{signal} e^{-\kappa Z_{noise}} \tag{A.4}$$

$$LIY_{signal} = LIY_m - \overline{I_m} * FT^{-1}\left[\sum_{n=1} C_n(\frac{\omega}{n}) G_m(\frac{\omega}{n})\right] \tag{A.5}$$

Method 3. Aliasing

$$A(\omega) = \frac{LI_{LIY_{cal}}(|\omega - \omega_{mod}|)}{LI_{LIY_{cal}}(|\omega|)} * \frac{1}{\overline{I_{cal}}} \ |\omega \ \leq 2 * \omega_{mod} \tag{A.6}$$

$$LIY_{alias}(\omega) = A(|\omega_{mod} - \omega|) * LIY_m(|\omega_{mod} - \omega|)$$

$$\begin{cases} RE(I_{alias}(\omega)) = \ RE(I_{alias}(-\omega)) \\ IM\big(I_{alias}(\omega)\big) = -IM(I_{alias}(-\omega)) \end{cases} \tag{A.7}$$

$$LIY_{alias}(t) = \ \overline{I_m} * FT^{-1}[LIY_{alias}(\omega)] \tag{A.8}$$

Appendix B

Synthetic Data Creation

Due to experimental an experimental setback it was not possible to collect data in the first half of this project. To remedy this fact a synthetic data creation program was created in Python. The purpose of this was to allow rapid testing of algorithm iterations with control over numerous variables to be tested. However, I did not utilize this program. I wrote the program to be able to create a data set similar to the test data that was collected in previous iterations of this project but allows the user to test particular functionality of the algorithm. For instance, in the case harmonic distortion, due to the nature by which data was previously collect, in steady-state, it would be difficult to discern whether frequency peaks are the result of integer multiples of a lower frequency or the result of the attenuation and phase shift of the fundamental frequency itself. To remedy this and test the capabilities of an algorithm to remove these distortions the program allows the user to manipulate the height and width of frequency peaks. For example, a definite harmonic response can be added to the realistic data set to understand the capabilities of the tested algorithm at removing those features.

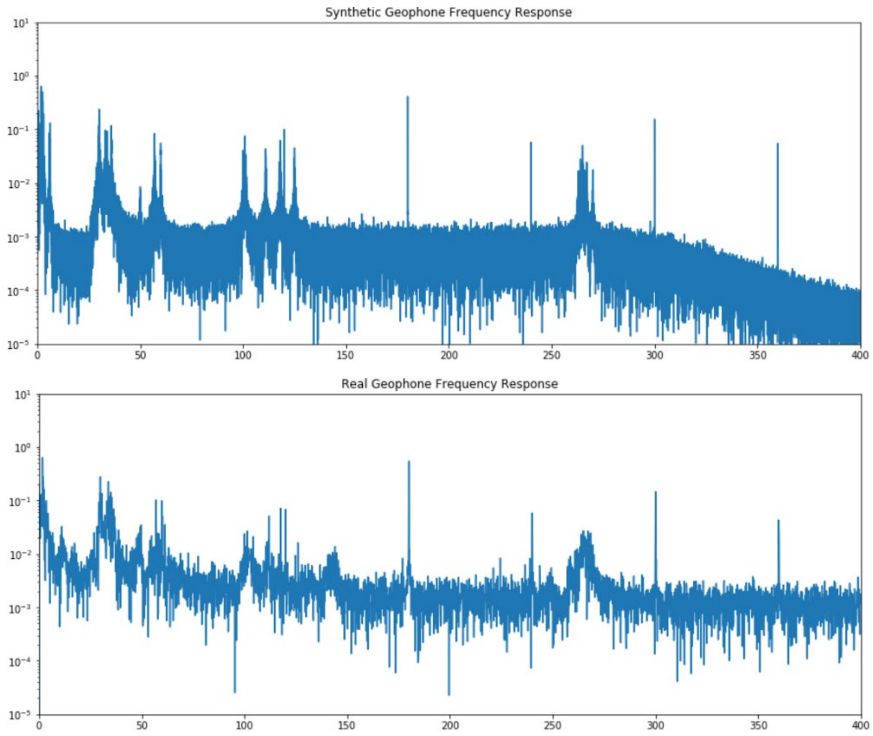


Figure B.1- The synthetic data represents previously collected data that can be generated and manipulated to test the abilities of the cancellation algorithm. This scheme is able to generate large amounts of data to test at ~1/10 the actual test time, depending on the number of unique features being rendered. In early stages of the algorithm this would have been useful in testing its performance, but I did not utilize it.

Appendix C

Data parsing

1. Frequency chirp isolation

An assumption that is made for the coefficient calibration is that the calibration data is collected only during the frequency chirp. This is not the case since there is an undefined amount of time between turning on the datalogger and starting the frequency sweep plus and undefined amount of time between

the frequency sweep ending and the datalogger being turned off. To ensure that the coefficients are accurately calibrated the calibration data set must be properly edited so that the input is solely the data collected during the frequency chirp. This is accomplished using a digital lockin calculation, utilizing Equation 2.5 but without integrating over the signal. When the integral is not performed the data that aligns with the reference signal will be at a constant dc offset. The location of the first and last frequency step are found by observing where the data is no longer constant. The beginning and ending of the frequency chirp are found by subtracting a step time from the first step location and adding a step time to the last step location, respectively. These locations and the form of the unintegrated lockin signal can be seen in Figure C.1.1. The code that generated this figure and found these locations can be found in Appendix D.15.

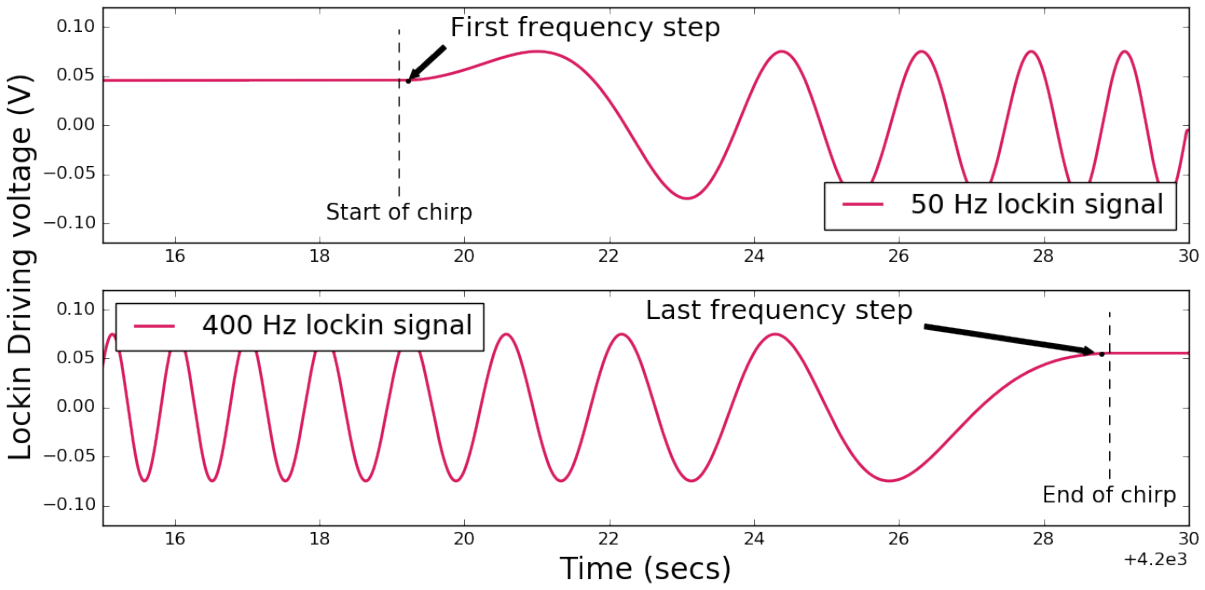


Figure C.1.1: Determining the beginning and end of the calibration data set. A digital lockin calculation was performed on the voltage driving the speaker to determine the beginning and end of the frequency chirp. Analyzing the beginning of the data set, using a reference frequency of 50 Hz, yields that the first frequency step occurs 19.22 seconds after the datalogger was turned on. Analyzing the end

of the data set, using a reference frequency of 400 Hz, yield that the last frequency step occurs 4228.79 seconds after the datalogger was turned on. This results in the beginning of the frequency chirp calculated to start 19.1 seconds after the datalogger was turned on and the end calculated at 4228.91 seconds after activation. *Data shown was collected at a 10 kHz sampling rate*

2. Sweep change identification

Separating the spectroscopic data into individual sweeps follows 1-2 steps, depending on the cancellation scheme that is being used. The different schemes are either the exact propagation method, where noise is incrementally estimated at the Z position of the tip to current data and finally to LIY data, or the direct propagation method, where noise is estimated directly from the geophone to current or LIY data. The first step in both schemes is separating the individual voltage sweeps in the data. This is done with a gradient mapping method that determines when the voltages resets from the top of the sweep to the bottom, this phenomenon is shown in Figure C.2.1. The code that determines these points can be found in Appendix D.16. This separation is all that is needed in the exact propagation scheme as it cancels noise from current data regardless of the I_0 value the data was collected at.

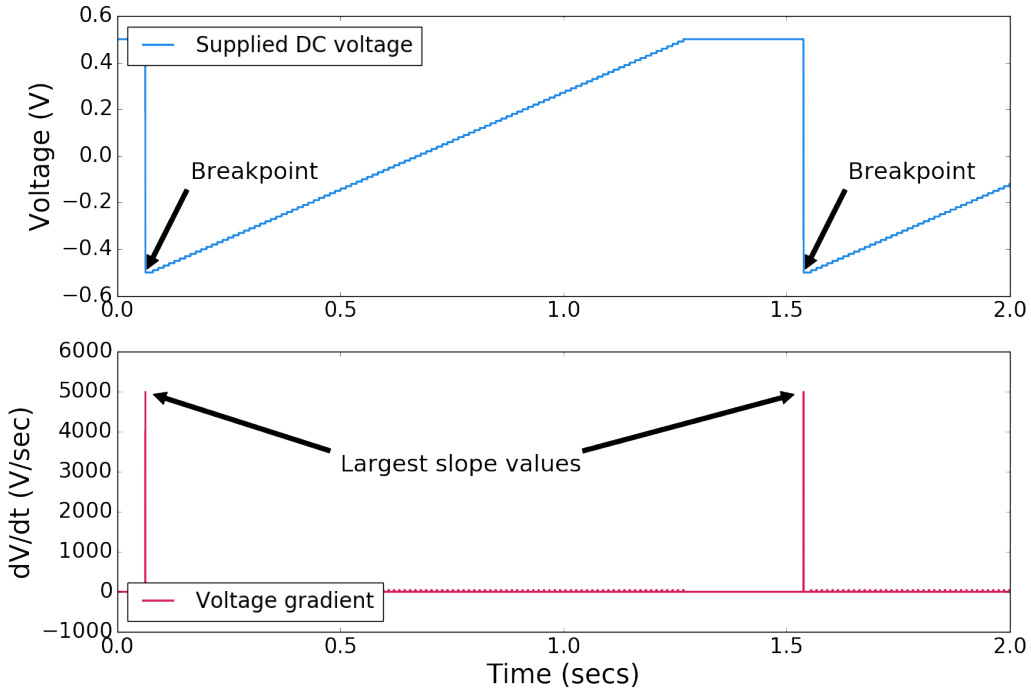


Figure C.2.1: Sectioning spectroscopic data with the voltage sweep data. The dc voltage supplied to the sample can be used to determine where the beginning and end of each sweep is. The voltage resets each time a sweep is performed (top) resulting in a large change of voltage that can be easily identified by calculating the magnitude of the voltage data gradient (bottom). *Data shown was collected at a 10 kHz sampling rate*

The second step, which is needed for the direct propagation scheme, is performing a similar gradient mapping method to the first on the now partitioned voltage data to determine where voltage steps during the sweep occur. These voltage steps range in magnitude between the amount the voltage is modulated at to 50 mV depending on the test. The determinations of these locations can be seen in Figure C.2.2. The code that determines these points can be found in Appendix D.16

.

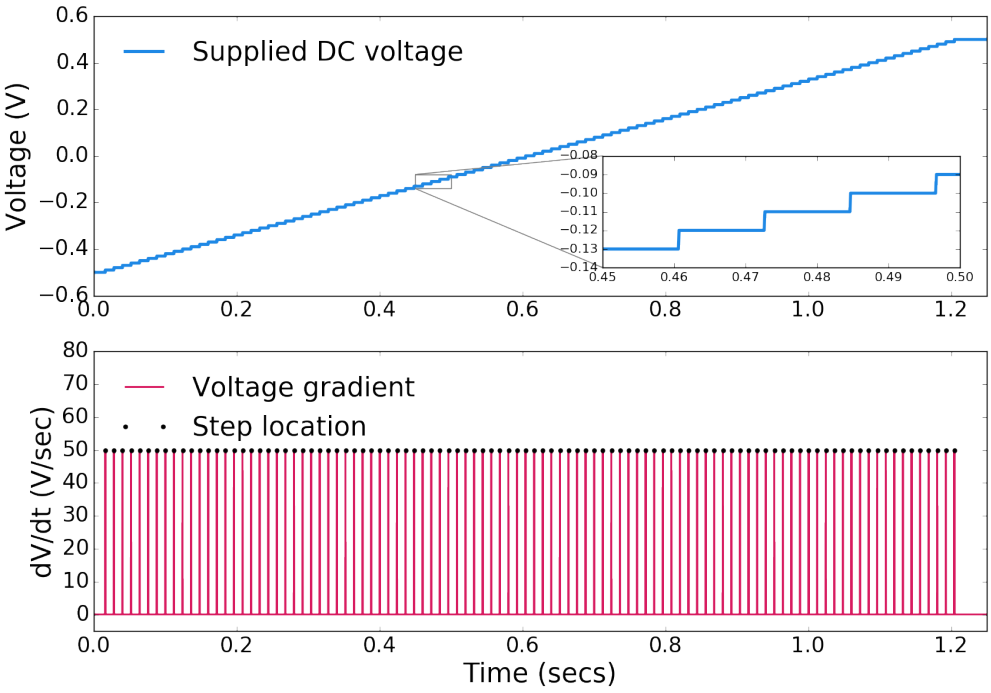


Figure C.2.2: Sectioning each step in the voltage sweep. Once the individual voltage sweeps are separated the location of each voltage step in those sweeps is found using another gradient map. The top plot shows a single -500 mV to 500 mV voltage sweep with a step size of 10 mV (step size seen in the inset plot of top). The large slope of these step changes allows them to be easily isolated from the rest of the data with a gradient map (bottom). *Data shown was collected at a 10 kHz sampling rate*

Appendix D

Python Code

1. Implementation of Equations 2.4, 2.8

```
#Use the sectioned calibration data to calculate the coefficients
#  with Equations 2.5-2.7
C_n, fftfreq = calibrate(geo_cal,z_cal)

#Calculate the Fourier transform of the measured geophone data
G_m = fft.fft(g_test)
omega = fft.fftfreq(len(g_test),1/10000) #find the corresponding frequencies

#Interpolate the geophone frequency signal
#  The harmonic calculation does not usually fall onto specific frequency bins,
#  the interpolation estimates the coefficient value if it falls between points
G_m = scipy.interpolate.interp1d(omega,G_m,kind='linear',fill_value='extrapolate')

z_noise = np.zeros(len(omega),dtype='complex128') #initialize noise array

#Use the coefficients and interpolated geophone signal to estimate z_noise
for i in arange(C_n[:]):
    n = i + 1
    #Sum the impact that each harmonic has on the signal (Equation 2.4)
    z_noise += C_n[i](omega/n)*G_m(omega/n)

#Find the signal by subtracting noise from measured data (Equation 2.8)
z_sig = z_test - fft.ifft(z_noise)
```

2. Implementation of Equations 2.5-2.7

```
def settling(w, mode = 'linear', fs = 10000, period=20, time = 0.05):
    '''
    Calculate the amount of time in each step that is needed for the filter
      and frequency signal to settle.
    In a 'dynamic' situation, where the time per step is proportional to the
      frequency and tested number of periods, this time is dependent on the
      frequency being tested. ***Preferred method
    In a 'linear' situation, where the time per step is equal at every frequency
      that is tested, this time also ends up being equal for every frequency
    Returns the number of bins for that time

    Inputs:
        w      - Required: float denoting reference frequency
        mode   - Optional: String for version determinization of
                           time calculation Options: 'dynamic', 'linear'
        fs     - Optional: int denoting sampling rate of data collection
        period - Optional: int determining number of periods for dynamic mode
        time   - Optional: float determining the settling time for linear mode

    Returns:
        set_time - int containing number of bins to wait before performing lockin
```

```
                calc
    '''
    set_time = 0 #set a default time if neither mode is chosen

    if mode == 'linear':
        set_time = time*fs #Set a standard time regardless of frequency
    if mode == 'dynamic':  #Set the time proportional to the frequency tested
        set_time = (period/w)*fs
    return int(set_time)

def lockin(data, freq, fs=10000):
    '''
    Calculate the magnitude and phase of a signal in the data with
    respect to the reference frequency
    *Equation 2.5

    Inputs:
        data - Required: array containing discrete sampled time series data
        freq - Required: float for reference frequency that needs to be isolated
                         from data
        fs   - Optional: int stating sampling rate of data collection

    Returns:
        LI_data - Complex number denoting magnitude and phase of isolated signal
    '''
    t = linspace(0,len(data)/fs, len(data))  # time array for generating the ref
                                             #  signal
    set_time = settling(freq)      #amount of time needed for the filter and
                                #  lockin amplifier to settle proportional to freq
    ref_sig = exp(-1j*freq*2*pi*t) #create the reference signal
    data_mix = data*ref_sig        #Mix the ref signal with the chopped data
    print(len(data_mix))
    #6 cascaded second order lowpass sections are used to isolate the DC component
    #     produced from the signal mixing with a cutoff frequency << w. This gives
    #     the equivalent of the fft bin value if a Fourier transform was
    #     able to be properly calculated.
    sos = signal.butter(6, freq, 'lowpass', fs=fs, output='sos')

    #A forward-backward digital filter using the cascaded sections
    #   results in filter with zero phase change
    data_filt = signal.sosfiltfilt(sos,data_mix)

    #In a discretely sampled series, the average of data points is equal to
    #   the integral over the same period
    #A time proportional to w at the beginning of the data is ignored for
    #   settling time of the filter and signal driven by the lockin amplifier
    LI_data = average(data_filt[set_time:])
    return LI_data

def coef(C,w,inpt,outpt,p1,p2,n,i,end,fs=10000):
    '''
    Calculate the response the output signal has to the input signal,
    isolated using a reference frequency with respect to 'w', also
    removing the impact that frequencies other than the harmonic response
    have on the output signal

    Inputs:
        C     - Required: 2D numpy array as the coefficient matrix
```

```
        w     - Required: float denoting the fundamental frequency
        intpt - Required: 1D numpy array used to calculate the
                          denominator of the transfer function (Geophone signal)
        outpt - Required: 1D numpy array used to calculate the
                          numerator of the transfer function (Z position, Current,
                          or LIY)
        p1    - Required: int representing the beginning of the calibration step
        p2    - Required: int representing the end of the calibration step
        n     - Required: int stating the harmonic of w that is being calculated
        i     - Required: int bin location of w in the coefficient matrix
        end   - Required: int/float ending frequency of calibration chirp
        fs    - Optional: int stating sampling rate of data collection

    Returns:
        coefficient - complex representation of how a signal is transferred from
                        the input to the output
    '''

    #If the harmonic freq. is above the Nyquist freq. do not try to calculate it
    if n*w >= fs/2:
        return 0 + 0*1j

    k = linspace(0,n-1,n).astype('int')  #Identify the integer harmonics below the
                                         #  calculated harmonic being found
    inpt = inpt[p1:p2]-mean(inpt[p1:p2]) #Remove the dc component from the
                                          #  calibration step
    outpt = outpt[p1:p2]                 #isolate the determined calibration step

    LI_out = lockin(outpt,n*w)    #Calculate the magnitude and phase of the signal
                                         #  with respect to the harmonic reference
                                         #  signal in output data
    LI_in = lockin(inpt,w)               #Same with the fundamental input data
    LI_sub = 0 + 0j                      #Initiate the interfering input signal
    #Iterate through each harmonic that is below the one being calculated
    #   Using them to make sure that the lockin signal is the result of harmonic
    #   distortion
    for sub in k:
        if sub == 0:  #If the first harmonic is being calculated there is not
                      #  other interference in the input signal
            continue
        if (n*w/sub) > end:
            return 0 + 0j
        LI_in_k = lockin(inpt,(n*w)/sub) #Calculate the mag and phase of signal
                                         #  with respect to the harmonics
        #Calculate the sum of interfering signals
        #  Estimated using the already defined coefficients
        LI_sub = LI_sub + C[int(n*i/sub),n-1]*LI_in_k

    coefficient = (LI_out-LI_sub)/LI_in #Return the calculated coefficient
    return coefficient

def calibrate(inpt,outpt,start=50,end=400,harmonics=5,steps=35000,fs=10000):
    '''
    Calibrates each coefficient using the already cut calibration data
    (cut to the beginning and end of the calibration chirp)
    *Equations 2.6-2.7

    Inputs:
```

```
        intpt    - Required: 1D numpy array used to calculate the
                           denominator of the transfer function (Geophone signal)
        outpt    - Required: 1D numpy array used to calculate the
                           numerator of the transfer function (Z position, Current,
                           or LIY)
        start   - Optional: int/float denoting the starting frequency of the chirp
        end      - Optional: int/float denoting the ending frequency of the chirp
        harmonics- Optional: number of harmonics/coefficients to be calculated
        steps    - Optional: number of frequency steps in the calibration chirp
        fs       - Optional: int stating sampling rate of data collection

    Returns:
        coef_mat - 1D/2D list of complex interpolations representing the response
                           of the output data to the input data during calibration
        fftfreq  - 1D numpy array of frequencies used for interpolation
    '''

    #This identifies the location of each change of frequency
    #  Since the step length was too small for proper calibration 30 steps
    #  are being averaged together, leading to some amount of error
    loc = linspace(0,len(t),int(steps/30)).astype('int')
    #Identify the reference frequency for each step to lock on to
    freq_map = linspace(start,end,len(loc))

    #Calculate the fft size that produces the same frequency resolution
    fftfreq = fft.fftfreq(int(fs/(freq_map[1]-start)),1/fs)

    #Initialize the coefficient matrix
    C = np.zeros((len(fftfreq),harmonics),dtype='complex128')

    #Calibrate the frequency response at each harmonic
    for n in arange(harmonics):
        prev = 0
        #Iterate through each frequency within the calibration bandwidth
        for w,point,i in zip(freq_map,loc[1:],arange(len(fftfreq))):
            #Harmonic response can only be found within the frequency bandwidth
            if (n+1)*w > end:
                continue
            i = i + 5000 #The calibration bandwidth does not start at 0 Hz
            #Call the function to calculate the coefficient
            C[i,n] = coef(C,w,inpt,outpt,prev,point,n+1,i,fs=fs)
            prev = point #Assign the beginning of the next frequency step
            print(fftfreq[i],end='\r') #Print status during calculation

    #Negative frequencies are the complex conjugate of the positive
    #   frequencies so the same is true of the coefficients
    C[int(len(C[:,0])/2):,:] = flip(conj(C[1:int(len(C[:,0])/2)+1]),axis=0)

    #Interpolate each set of coefficients and save in a list
    coef_mat = harmonics*[0]
    for n in arange(harmonics):
        coef_mat[n] = scipy.interpolate.interp1d(fftfreq, C[:,n],
                                          kind='linear', fill_value='extrapolate')

    return coef_mat, fftfreq

C, freqs = calibrate(geo_cal,z_cal) #Calibrate the coefficients
```

3. Implementation of Equations 2.13-2.14

```
#Fit the Z spectroscopy data to find the kappa constant
fit = polyfit(z_spec[:,0],log(abs(z_spec[:,1])),1)
kappa = fit[0] #kappa represents the semi-logarithmic slope value of the
               #  recorded current as the tip is withdrawn from the sample

#Current noise is estimated
cur_noise = exp(-kappa*(z_noise-mean(z_noise)))

#The current is removed from the measured data through division
#  *Dividing two exponents with the same base results in the
#    exponents just being subtracted
#  **This also means that the noise is removed from the measured data
#     independent of the initial current value in the measured data
#  ***This is a particularly useful trait in spectroscopic cancellation
#     where the initial current value changes with the sweeping voltage
cur_sig = cur_m/cur_noise #Equation 2.14
# *Equation 2.17 is also applied to cancel LIY data by estimating
#    the amount of noise in LIY and dividing it by the measured LIY
```

4. Kappa calculation and figure generation

```
#load Z-spectroscopy data that was recorded
folder_loc = '2020-03-13/'
file_loc = ['kappa00002.dat','kappa00003.dat','kappa00004.dat',
            'kappa00005.dat','kappa00006.dat']
#Initialize the fit parameters
#  *It is a 1st degree fit so only slope and dc offset are needed
z = np.zeros((5,102))
cur = np.zeros((5,102))
fit = np.zeros((5,2))
#Iterate through each file name and load the corresponding data
#  Each file has 2 columns, the Z position and the corresponding current value
for i,name in enumerate(file_loc):
    spec = np.loadtxt(folder_loc+name,skiprows=126)
    #Separate the columns of each data set
    z[i,:] = spec[:,0]
    cur[i,:] = spec[:,1]
    #Fit the log of the current data with a 1-degree polynomial
    fit[i,:] = np.polyfit(spec[:40,0], log10(abs(spec[:40,1])),1)
#Average together all the fit parameters
fit = mean(fit, axis=0)
#kappa is equal to the slope of the fit
kappa = fit[0]


fig,ax = subplots(1,1)
figsize(10,10)
fitted = np.zeros(102) #this is the length of the loaded files
for i in arange(5): #Iterate through each set of data, plotting it
    #Create an object to fit the log of the current data
    #Kappa represents the fit of the linear portion of the data
    #  which in this data is only at the beginning
    p = np.poly1d(np.polyfit(z[i,:30], log10(abs(cur[i,:30])),1))
```

```
    #plot the log of the current data against the z position of the tip
    ax.plot(1e9*(z[i,:]-z[i,0]),log10(abs(cur[i,:])),'-')
    fitted += p(z[i,:]) #sum the fitted data sets together
#plot the average fit to the data sets
ax.plot(1e9*(z[1,:]-z[1,0]),fitted/5,'r',linewidth=3,label='Linear Fit')
xlim(0,0.2)
ylim(-35,-20)
#Print the corresponding kappa value on the plot
ax.text(0.14,-22.4,'$\u03BA = -3.47 * 10^{10}$',fontsize=18)
xlabel('Distance (nm)',size=20)
ylabel('Log Current log(A)',size=20)
legend(loc=1,fontsize=15)
plt.show()
```

5. Implementation of Equations 2.11-2.12

```
def fit_cancel(array, n=2):
    '''
    **This function was modified from code written by Albert Chen
    Remove drift with a n-degree polynomial function fitted to the data

    Inputs:
        array - Required: 1D numpy array to be cancelled
        n     - Optional: degree of polynomial used for fitting

    Returns:
        Zero centered numpy array, same length as 'array'
    '''
    #Create a linearly separated set of point to fit array with
    t = np.linspace(0,len(array),len(array))
    #Fit array using the numpy polyfit function and implement
    # the resulting coefficients into a polynomial object
    fit = poly1d(np.polyfit(t,array,n))
    #Calculate the corrective array using the fit object
    corrective = fit(t)
    #Return the corrected data
    return array - corrective

def segment(data,n,o,b=5):
    '''
    This function segments the data into predetermined pieces

    Inputs:
        data - Required: 1D numpy array of data to be segmented
        n    - Required: length of segment data is pieced into
        o    - Required: length of overlap between segments
        b    - Optional: Beta parameter used by the Kaiser function.
                    The default = 5 makes it similar to a Hamming window

    Returns:
        seg  - 2D array size(data-o//n-o,n) of the input data
    '''
    #Initialize a 2D array with the number of rows being how many segments
    #   can be created from data of length 'n' and columns of length 'n'
    seg = np.zeros(((len(data)-o)//(n-o),n),dtype=np.complex)
    temp = np.copy(data)
    i = 0
    #Parse through data, separating the segments into different rows
```

```
    #This does not utilize all the data, only up length divisible by n
    #Break out of the while loop once only the surplus data remains
    while len(temp) >= n:
        #Remove any dc offset or 2nd degree polynomial drift
        #  from the segmented data and save it to the designated row
        seg[i,:] = fit_cancel(temp[:n])
        i += 1 #iterate to the next row
        temp = temp[n-o:] #Redefine the data set, removing the last
                          #  segmented piece of data minus the overlap
    return seg

def avg_fft(data,n,o,b=5,fs=10000):
    '''
    The Fourier transform of the data, in entirety, is quite noisy.
    This function uses the segment function to break the data into manageable
pieces,
    averaging the Fourier transform of each segment together to reduce that noise.
    This reduces incoherent signals and random fluctuations in coherent ones.

    Inputs:
        data - Required: 1D numpy array of data to be segmented
        n    - Required: length of segment data is pieced into
        o    - Required: length of overlap between segments
        b    - Optional: Beta parameter used by the Kaiser function.
                    The default = 5 makes it similar to a Hamming window
        fs   - Optional: int stating sampling rate of data collection

    Returns:
        seg  - fft of length n
        freqs- frequencies the make up the fft
    '''
    #Call the segment function to parse through data
    seg = segment(data,n,o,b)
    #Calculate the Fourier transform of each row in the 2D array
    seg = fft.fft(seg,axis=1)
    #Average the Fourier transforms together
    seg = mean(seg,axis=0)
    #Calculate the corresponding frequencies
    freqs = fft.fftfreq(len(seg),1/fs)
    return seg,freqs

def cal_alias(data,w_mod,n,o,b=5,fs=10000):
    '''
    Calibrate the transfer function needed to remove spectroscopic
      aliasing from current and LIY data. This scheme is applied using
      data collected while the system is in steady state and is applied
      in parallel to the vector cancellation scheme. It also is only
      needed for vector cancellation when the 'Exact Propagation' method
      is used (G-Z-I-LIY). When the 'Direct Propagation' method is used
      (G-I or G-LIY) aliasing is accounted for, though not 'properly'.

    Inputs:
        data  - Required: 1D array containing the current or LIY calibration data
        w_mod - Required: the frequency the voltage bias was modulated at
        n     - Required: length of segment data is pieced into
        o     - Required: length of overlap between segments
        b     - Optional: Beta parameter used by the Kaiser function.
                    the default = 5 makes it similar to a Hamming window
```

```
        fs    - Optional: int stating sampling rate of data collection

    Returns:
        A     - Interpolated object representing the attenuation and phase shift
                by frequencies shifted due to the modulation of the voltage bias
        freqs - The frequencies by which the transfer function A was calibrated
    '''
    #Initialize the array as a complex set of numbers
    #   so, it saves any phase information as well as magnitude
    A = np.zeros(len(n),dtype='complex128')
    #Calculate the Fourier transform of the data and clean
    #  it up with some averaging
    fft_dat,freqs = avg_fft(data,n,o,b=b)
    w_alias = int(2*w_mod/freqs[1]) #indicates the range of the aliasing effect
    #Iterate through positive frequencies up to 2*w_mod and calculate
    #  the response that the frequencies about w_mod have to detected frequencies
    #  less than w_mod. *Implementation of Equation 2.13
    for i,frq in enumerate(freqs[:w_alias]):
        A[i] = fft_dat[absolute(i-w_alias)]/fft_dat[i]
    #Interpolate the transfer function so that frequencies called between
    #  bins can be estimated
    A = scipy.interpolate.interp1d(freqs,A,kind='linear',fill_value='extrapolate')

    return A, freqs


#Calibrate a transfer function between the vibration frequency peaks and
#   the corresponding aliased peaks
A,__ = cal_alias(cur_cal,1000,200000,50000,b=5,fs=10000)

#Take the Fourier transform of the measured current data
fft_cur_m = fft.fft(cur_test)

#Calculate the corresponding frequencies
freqs = fft.fftfreq(len(cur_test),1/10000)
w_alias = int(2*w_mod/freqs[1]) #indicates the range of the aliasing effect

#Initialize the array for the aliased noise
cur_alias = np.zeros(len(cur_test),dtype='complex128')

#Iterate up to 2*w_mod to estimate the amount of noise that is the result
#   of the aliasing effect *Implementation of Equation 2.11
for i in arange(w_alias):
    cur_alias[i] = A(absolute(freqs[w_alias-i]))*fft_cur_m[absolute(w_alias-i)]

#Ensure that the positive and negative frequencies of the aliased noise are
#   complex conjugates *Following the conditions of Equation 2.11
cur_alias[int(len(cur_alias)/2):] = np.flip(conj(
       cur_alias[:int(len(cur_alias)/2)]))
#Move the aliased noise back into the time domain *Equation 2.12
cur_alias = fft.ifft(cur_alias)
```

6. Implementation of Equations 2.15-2.16

```
rep = fs/w_mod #Number of points that are integrated over
#This calculation replicates the lock in integral
liy_noise = repeat(mean(cur_noise.reshape(-1,rep),axis=1),rep)
```

```
#Equation 2.16
liy_sig = liy_test/liy_noise
```

7. Generation of Figure 3.5-3.7

```
#Use the program Albert Chen wrote to calculate the transfer function for
#  the vector cancellation scheme and use the geophone test data to
#  estimate the noise
s,e = 3000000,3800000
g_cal,z_cal,g_test,z_test = geo_cal[:s],tip_cal[:s],geo_cal[s:e],tip_cal[s:e]

import cancelv3 as cancel
tf = cancel.Cancel(g_cal-mean(g_cal),z_cal,1/10000,p=[200000,50000,5])
z_noise_vec  = tf.create_drive(g_test-mean(g_test))
z_sig_vec = z_test - z_noise_vec #Subtract the noise from the measured tip data

#Figure 3.5
#Plot the raw and cancelled data sets
#  *The '-5' is used to center the signal about zero pm
#  ** The '-321' is used to center the displayed signal about 0 secs
plot(t_cal[start:end]-321,1e12*(z_test-mean(z_test))-5,
     '#1e88e5',linewidth=2,label='Raw')
plot(t_cal[start:end]-321,1e12*(z_sig.real-mean(z_sig.real))-5,
     '#d81b60',linewidth=2,label='Matrix cancelled')
plot(t_cal[start:end]-321,1e12*(z_sig_vec.real-mean(z_sig_vec.real))-5,
     '#ffc107',linewidth=2,label='Vector cancelled')
xlim(0,0.5)
ylim(-12,12)
tick_params(labelsize=20)
xlabel('Time (secs)',size=25)
ylabel('Tip position (pm)',size=25)
legend(loc=1,fontsize=20)

#_____________________________New Cell_____________________________
#Figure 3.6
#Calculate the Fourier transform of the raw and cancelled signals
n = 100000
o = 25000
b = 5
z_fft,freqs = avg_fft(z_test,n,o,b)
z_mat_fft,freqs_m = avg_fft(z_sig,n,o,b)
z_vec_fft,freqs_v = avg_fft(z_sig_vec,n,o,b)

#Plot all three data sets on a logarithmic plot
semilogy(freqs[:int(len(z_fft)/2):10],
       1e12*absolute(z_fft[:int(len(z_fft)/2):10])/n,
       '#1e88e5',label='Raw',linewidth=2)
semilogy(freqs_m[:int(len(z_mat_fft)/2):10],
       1e12*absolute(z_mat_fft[:int(len(z_mat_fft)/2):10])/n,
       '#d81b60',label='Matrix Cancelled',linewidth=2)
semilogy(freqs_v[:int(len(z_vec_fft)/2):10],
       1e12*absolute(z_vec_fft[:int(len(z_vec_fft)/2):10])/n,
       '#ffc107',label='Vector Cancelled',linewidth=2)

xlim(50,400)
ylim(1e-5,1e1)
legend(loc=0,fontsize=20)
```

```
tick_params(labelsize=20)
xlabel('Frequency (Hz)',size=25)
ylabel('Z ASD (pm)',size=25)

#_____________________________________New Cell_____________________________________
#Figure 3.7
#Calculate the Fourier transform of the raw and cancelled signals
n = 200000
o = 50000
b = 5
z_fft,freqs = avg_fft(1e12*z_test,n,o,b)
z_noise_fft,freqs_n = avg_fft(1e12*fft.ifft(z_noise),n,o,b)
z_noise_vec_fft,freqs_nv = avg_fft(1e12*(z_noise_vec),n,o,b)
z_fft,z_noise_fft,z_noise_vec_fft = (
    1e12*z_fft/n,1e12*z_noise_fft/n,1e12*z_noise_vec_fft/n)

freq_loc = 2480  #The bin number for the frequency by which i am specifically
                 #   calculating the phase of (124 Hz)

deg_sign= u'\N{DEGREE SIGN}' #This is just the code to create a degree sign string
#Create strings to display on the plot
#  Calculating the phase of signal at freq_loc
raw_phase = 'Raw phase: '+ str(around(180*angle(
    z_fft[freq_loc])/(2*pi))) +deg_sign
mat_phase = 'Matrix phase: '+ str(around(180*angle(
    z_noise_fft[freq_loc])/(2*pi))) + deg_sign
vec_phase = 'Vector phase: '+ str(around(180*angle(
    z_noise_vec_fft[freq_loc])/(2*pi))) + deg_sign

#Plot the fft and strings on a loglog plot
fig,ax = subplots(1,1)
figsize(10,7)
ax.semilogy(freqs[:int(len(z_fft)/2):10],
       absolute(z_fft[:int(len(z_fft)/2):10]),
       '#1e88e5',label='Raw',linewidth=2)
#This just adds an arrow pointed at the freq_loc location
ax.annotate(raw_phase, xy=(freqs[freq_loc],abs(z_fft[freq_loc])),
            xytext=(freqs[freq_loc-1200],10*abs(z_fft[freq_loc])),
            fontsize=15,arrowprops=dict(
                width=5,headwidth=12,headlength=12,edgecolor='w',
                facecolor='#1e88e5', shrink=0.04)
              )
ax.semilogy(freqs_n[:int(len(z_noise_fft)/2):10],
       absolute(z_noise_fft[:int(len(z_noise_fft)/2):10]),
       'k',label='Matrix Estimation',linewidth=2,alpha=0.8)
ax.annotate(mat_phase, xy=(freqs_n[freq_loc],abs(z_noise_fft[freq_loc])),
            xytext=(freqs_n[freq_loc+500], 2*abs(z_noise_fft[freq_loc])),
            fontsize=15,arrowprops=dict(
                width=5,headwidth=12,headlength=12,edgecolor='w',
                facecolor='k', shrink=0.04)
              )
ax.semilogy(freqs_nv[:int(len(freqs_nv)/2):10],
       absolute(z_noise_vec_fft[:int(len(z_noise_vec_fft)/2):10]),
       'g',label='Vector Estimation',linewidth=2,alpha=0.6)
ax.annotate(vec_phase, xy=(freqs_nv[freq_loc],abs(z_noise_vec_fft[freq_loc])),
            xytext=(freqs_nv[freq_loc-1200], 2*abs(z_noise_vec_fft[freq_loc])),
            fontsize=15,arrowprops=dict(
                width=5,headwidth=12,headlength=12,edgecolor='w',
```

```
                facecolor='g', shrink=0.04)
              )
ax.legend(loc=0,fontsize=20)
ax.tick_params(labelsize=20)
ax.set_xlabel('Frequency (Hz)',size=25)
ax.set_ylabel('Z ASD (pm)',size=25)
ax.set_xlim(50,400)
ax.set_ylim(1e-5,1e1)

#Time trace of noise plot
plot(t_cal[s:e]-300,1e12*(z_test-mean(z_test)),
     '#1e88e5',label='Raw',linewidth=2)
plot(t_cal[s:e]-300,1e12*fft.ifft(z_noise),
     'k',label='Matrix Estimation',linewidth=2)
plot(t_cal[s:e]-300,1e12*z_noise_vec,
     'g',label='Vector Estimation',linewidth=2)
xlim(0.5,0.6)
ylim(-150,150)
legend(loc=1,fontsize=20)
tick_params(labelsize=15)
xlabel('Time (secs)',size=25)
ylabel('Tip position (pm)',size=25)
savefig('2020-03-13 Noise phase issue',transparent=True)
```

8. Z position RMS reduction calculation

```
#Segment the raw and cancelled data into sections of 5 seconds
n=50000

#Use the segment function to do this with overlap = 0
seg_raw,seg_mat,seg_vec = \
segment(z_test,n,0),segment(z_sig,n,0),segment(z_sig_vec,n,0)

#Calculate the rms value of each of the 5 sec segments
rms_raw,rms_mat,rms_vec = \
zeros(len(seg_raw[:,0])),zeros(len(seg_raw[:,0])),zeros(len(seg_raw[:,0]))
for i in arange(len(seg_raw[:,0])):
    rms_raw[i] = absolute(sqrt(mean((seg_raw[i,:]-mean(seg_raw[i,:]))**2)))
    rms_mat[i] = absolute(sqrt(mean((seg_mat[i,:]-mean(seg_mat[i,:]))**2)))
    rms_vec[i] = absolute(sqrt(mean((seg_vec[i,:]-mean(seg_vec[i,:]))**2)))

#Calculate the percent reduction that matrix and vector schemes achieve
canc_mat = 100-100*rms_mat.astype('float')/rms_raw.astype('float')
canc_vec = 100-100*rms_vec.astype('float')/rms_raw.astype('float')

print('Mean Matrix:',mean(canc_mat))
print('STD Matrix: ',std(canc_mat))
print('Mean Vector:',mean(canc_vec))
print('STD Vector:',std(canc_vec))
```

9. Generation of Figure 4.1

```
from mpl_toolkits.axes_grid.inset_locator import (
    inset_axes, InsetPosition,mark_inset)
```

```
prev,count = 0,0
avg_swp = np.zeros((swp_pnt[1]-swp_pnt[0]),dtype='float64')
#Sum through each voltage sweep and sum them together
for brk in swp_pnt:
    avg_swp += cur_cut[prev:brk]
    count += 1
    prev=brk
#Divide by the number of sweeps to get the average
avg_swp = avg_swp/count

sweep = 235 #detail the sweep number to display
delay=10 #a delay is necessary to componsate for time
         #  effects between the geophone/Z and current data
fig,ax = subplots(1,2)
figsize(25,12)
#Use sweep to determine which sweep to display
loc1 = swp_pnt[sweep-1]
loc2 = swp_pnt[sweep]

#Calculate the average value of current over each voltage step
#  to determine how current changes with each change of voltage
cur_IV = np.mean(cur_cut[:7356720].reshape(-1, 120), axis=1)
mat_IV = np.mean((cur_cut[:7356720]/
                  cur_mat_n[:7356720]).reshape(-1, 120), axis=1)
vec_IV = np.mean((cur_cut[:7356720]/
                  cur_vec_n[:7356720]).reshape(-1, 120), axis=1)

ax[1].plot(bias_cut[loc1:loc2-delay:120],
     1e9*cur_IV[int(loc1/120):int((loc2-delay)/120)],
     '#1e88e5',label='Raw',linewidth=2)
ax[1].plot(bias_cut[loc1:loc2-delay:120],
     1e9*mat_IV[int(loc1/120):int((loc2-delay)/120)],
     '#d81b60',label='Matrix Cancelled',linewidth=2)
ax[1].plot(bias_cut[loc1:loc2-delay:120],
     1e9*(vec_IV[int(loc1/120):int((loc2-delay)/120)]),
     '#ffc107',label='Vector Cancelled',linewidth=2)
ax[1].plot(bias_cut[loc1:loc2-120:120],1e9*avg_IV,
     label='Average 499 sweeps',linewidth=2)
ax[1].legend(loc=2, fontsize = 25,fancybox=True, framealpha=0)
ax[1].set_xlabel('Voltage (mV)',size=33)
ax[1].tick_params(labelsize=20)
ax[1].set_xlim(-0.5,0.5)

ax[0].plot(t[loc1:loc2-delay]-t[loc1],1e9*cur_cut[loc1:loc2-delay],
         '#1e88e5',label='Raw')
ax[0].plot(t[loc1:loc2-delay]-t[loc1],1e9*(
    cur_cut[loc1:loc2-delay]/cur_mat_n[loc1+delay:loc2]),
         '#d81b60',label='Matrix Cancelled')
ax[0].plot(t[loc1:loc2-delay]-t[loc1],1e9*(
    cur_cut[loc1:loc2-delay]/cur_vec_n[loc1+delay:loc2]),
         '#ffc107',label='Vector Cancelled')
ax[0].plot(t[loc1:loc2-delay]-t[loc1],1e9*avg_swp[:-delay],
         'b',label='Averaged 499 sweeps',linewidth=2)
ax[0].grid(False)
ax[0].tick_params(labelsize=20)
ax[0].set_xlabel('Time/Voltage (secs/mV)',size=35)
ax[0].set_ylabel('Current (nA)',size=35)
```

```
ax[0].set_xlim(0,1.25)
ax[0].set_ylim(-1,1.2)
ax2 = plt.axes([0,0,1,1])
ip = InsetPosition(ax[0], [0.55,0.06,0.4,0.4])
ax2.set_axes_locator(ip)
mark_inset(ax[0], ax2, loc1=2, loc2=4, fc="none", ec='0.5')

ax2.plot(t[loc1:loc2-delay]-t[loc1],1e9*cur_cut[loc1:loc2-delay],
         '#1e88e5',label='Raw')
ax2.plot(t[loc1:loc2-delay]-t[loc1],1e9*(
    cur_cut[loc1:loc2-delay]/cur_mat_n[loc1+delay:loc2]),
         '#d81b60',label='Matrix Cancelled')
ax2.plot(t[loc1:loc2-delay]-t[loc1],1e9*(
    cur_cut[loc1:loc2-delay]/cur_vec_n[loc1+delay:loc2]),
         '#ffc107',label='Vector Cancelled')
ax2.plot(t[loc1:loc2-delay]-t[loc1],1e9*avg_swp[:-delay],
         'b',label='Avg 499 sweeps',linewidth=2)
ax2.grid(False)
ax2.tick_params(labelsize=18)
ax2.set_xlim(0.9,1.2)
ax2.set_ylim(0.4,1)
ax[0].legend(loc=2, fontsize = 25,fancybox=True, framealpha=0)
```

10. Current sweep RMS reduction calculation

```
def rms(array,n=1):
    return sqrt(mean((fit_cancel(array,n))**2))

dt = t[1] #The change in time during data collection
grad = np.gradient(bias_cut,dt)
delay = 10
#Implement Equation 2.19 to cancel noise
cur_mat = cur_cut[:-delay]/cur_mat_n[delay:]
cur_vec = cur_cut[:-delay]/cur_vec_n[delay:]
#Iterate through each sweep
prev = 0
start = time.time()
rms_cur = np.zeros((len(swp_pnt)))
rms_cur_mat = np.zeros((len(swp_pnt)))
rms_cur_vec = np.zeros((len(swp_pnt)))
for idx_brk,brk in enumerate(swp_pnt):
    #find the step points in the
    grad_brk = grad[prev:brk-1000]
    #Determine the top 200 gradient values in this sweep
    #  *In this test data there were only about 100 voltage
    #    steps. With each step being captured by ~2 points.
    #  **Based on the step magnitude and number this is subject
    #    to change
    step_pnt = np.flip(np.argsort(absolute(grad_brk)))[:300]

    count = 0
    #Iterate through all 200 gradient values
    for j in step_pnt:
        #Determine if they meet the expected threshold
        if absolute(grad_brk[j]) > 20:
```

```
            count = count+1
        #Redefine the top count gradient values
        step_pnt = np.flip(np.argsort(absolute(grad_brk)))[:count]
        #Iterate through the determined points and remove any neighboring points
        for idx,point in enumerate(step_pnt):
            #Set the value of any location within 10 pins of point to 0
            step_pnt[np.where(absolute(step_pnt-point)<10)] = 0
            #This also sets the point to 0, so reestablish its value
            step_pnt[idx] = point
        #Remove any 0 values in the array
        step_pnt = np.sort(step_pnt[step_pnt != 0])
        step_pnt = prev + step_pnt

        I_n,reps = np.zeros(len(step_pnt)),np.zeros(len(step_pnt))
        #Calculate the average current value at each voltage step
        pv=prev
        for idx,point in enumerate(step_pnt):
            I_n[idx] = mean(cur_cut[pv:point])
            reps[idx] = point-pv
            pv = point
        reps[-1] = reps[-1] +1

        #Repeat the calculated numbers to ensure that it
        #  is the same shape as the sweep
        I_n = repeat(I_n,reps.astype('int'))

        rms_cur[idx_brk] = rms(cur_cut[prev:point]-I_n[:-1])
        rms_cur_mat[idx_brk] = rms(cur_mat[prev:point]-I_n[:-1])
        rms_cur_vec[idx_brk] = rms(cur_vec[prev:point]-I_n[:-1])
        print(idx_brk,end='\r')
        prev = brk

mat_canc = 100-100*(rms_cur_mat/rms_cur)
vec_canc = 100 - 100*(rms_cur_vec/rms_cur)
print('IV Mat mean:', mean(mat_canc))
print('IV Mat std:', std(mat_canc))
print('IV Vec mean:', mean(vec_canc))
print('IV Vec std:', std(vec_canc))
```

11. Generation of Figure 4.3-4.4

```
#Figure 4.3
#noise calculation for the matrix and vector estimation
cur_mat_n = exp(-kappa*(fft.ifft(z_noise)
                        -mean(fft.ifft(z_noise))))
cur_vec_n = exp(-kappa*(z_sp_n_vec-mean(z_sp_n_vec)))

#average each sweep together to find the 'Ground Truth'
prev,count = 0,0
avg_liy = np.zeros((swp_pnt[1]-swp_pnt[0]),dtype='float64')
#Sum through each voltage sweep and sum them together
for brk in swp_pnt:
    avg_liy += LIY_cut[prev:brk]
    count += 1
    prev=brk
#Divide by the number of sweeps to get the average
avg_liy = avg_liy/count
```

```
sweep = 219 #detail the sweep number to display
#Use sweep to determine which sweep to display
loc1 = swp_pnt[sweep-1]
loc2 = swp_pnt[sweep]

#Implement Equation 2.14
liy_mat = (LIY_cut/cur_mat_n)
liy_vec = (LIY_cut/cur_vec_n)
#Implement Equation 2.14
mat_rep = repeat(np.mean(
    liy_mat[:7356740].reshape(-1, 10), axis=1),10)
vec_rep = repeat(np.mean(
    liy_vec[:7356740].reshape(-1, 10), axis=1),10)


plot(bias_cut[loc1:loc2-delay:120],
     1e12*LIY_cut[loc1:loc2-delay:120],
     '#1e88e5',label='Raw',linewidth=2)
plot(bias_cut[loc1:loc2-delay:120],
     1e12*mat_rep[loc1:loc2-delay:120],
     '#d81b60',label='Matrix Cancelled',linewidth=2)
plot(bias_cut[loc1:loc2-delay:120],
     1e12*vec_rep[loc1:loc2-delay:120],
     '#ffc107',label='Vector Cancelled',linewidth=2)
plot(bias_cut[loc1:loc2-delay:120],
     1e12*avg_liy[::120],
     'b',label='Avg 500 sweeps',linewidth=2)

legend(loc=0,fontsize=20)
xlabel('Voltage (V)',size=30)
ylabel('dI/dV (nS)',size=30)
tick_params(labelsize=15)

xlim(-0.5,0.5)
ylim(-5,40)

#_____________________________New Cell_______________

#Figure 4.4

liy_fft = np.zeros(len(LIY_cut[loc1:loc2-2580]),dtype='complex128')
liy_mat_fft = np.zeros(len(LIY_cut[loc1:loc2-2580]),dtype='complex128')
liy_vec_fft = np.zeros(len(LIY_cut[loc1:loc2-2580]),dtype='complex128')
freqs = fft.fftfreq(len(liy_mat_fft),1/10000)
prev = 0
#average the fft taken over each sweep to reduce noise
for pnt in swp_pnt[:499]:
    x = (pnt-prev) -len(liy_fft)
    liy_fft = liy_fft + fft.fft(1e12*LIY_cut[prev:pnt-x])
    liy_mat_fft = liy_mat_fft + fft.fft(1e12*mat_rep[prev:pnt-x])
    liy_vec_fft = liy_vec_fft + fft.fft(1e12*vec_rep[prev:pnt-x])
liy_fft = (liy_fft/499)/len(liy_fft)
liy_mat_fft = (liy_mat_fft/499)/len(liy_mat_fft)
liy_vec_fft = (liy_vec_fft/499)/len(liy_vec_fft)
avg_liy_fft = fft.fft(1e12*avg_liy)/len(avg_liy)
```

```
# freqs = fft.fftfreq(len(liy_calc_fft),1/10000)
# print(np.where(abs(freqs-62)<1),print(absolute(liy_calc_fft[75:77])**2))
semilogy(freqs[:int(len(freqs)/2):2],
         absolute(liy_fft[:int(len(freqs)/2):2]),
         '#1e88e5',label='Raw',linewidth=2)
semilogy(freqs[:int(len(freqs)/2):2],
         absolute(liy_mat_fft[:int(len(freqs)/2):2]),
         '#d81b60',label='Matrix Cancelled',linewidth=2)
semilogy(freqs[:int(len(freqs)/2):2],
         absolute(liy_vec_fft[:int(len(freqs)/2):2]),
         '#ffc107',label='Vector Cancelled',linewidth=2)
semilogy(freqs[:int(len(freqs)/2):2],
         absolute(avg_liy_fft[:int(len(freqs)/2):2]),
         'b',label='Avg 500 Sweeps',linewidth=2)
xlim(50,400)
# ylim(1e-3,1e2)
ylim(1e-3,1e0)
legend(loc=0,fontsize=20)
tick_params(labelsize=15)
xlabel('Frequency (Hz)',size=30)
ylabel('dI/dV (nS)',size=30)
```

12. LIY deviation from avg calculation

```
#Implement Equation 2.14
liy_mat = (LIY_cut/cur_mat_n)
liy_vec = (LIY_cut/cur_vec_n)
#Implement Equation 2.14
mat_rep = repeat(np.mean(
    liy_mat[:7356740].reshape(-1, 10), axis=1),10)
vec_rep = repeat(np.mean(
    liy_vec[:7356740].reshape(-1, 10), axis=1),10)
#Creates arrays of the shape for dI/dV vs V
liy_calc = np.zeros((len(swp_pnt),123))
liy_mat_calc = np.zeros((len(swp_pnt),123))
liy_vec_calc = np.zeros((len(swp_pnt),123))

prev = 0
for idx,sweep in enumerate(swp_pnt):
    loc1 = prev
    loc2 = sweep
    liy_calc[idx,:] = LIY_cut[loc1:loc2-delay:120] - avg_liy[::120]
    liy_mat_calc[idx,:] = mat_rep[loc1:loc2-delay:120] - avg_liy[::120]
    liy_vec_calc[idx,:] = vec_rep[loc1:loc2-delay:120] - avg_liy[::120]
    prev=sweep
print(shape(std(1e12*liy_calc,axis=0)))
liy_std = (std(1e12*liy_calc,axis=0))
mat_std = (std(1e12*liy_mat_calc,axis=0))
vec_std = (std(1e12*liy_vec_calc,axis=0))
mat_canc = 100-100*abs(mat_std/liy_std)
vec_cenc = 100-100*abs(vec_std/liy_std)
print('LIY std from avg: ',mean(liy_std),'nS')
print('Mat std from avg: ',mean(mat_std),'nS')
print('Vec std from avg: ',mean(vec_std),'nS')
```

```
print('Mat reduction: ', mean(mat_canc[2:]))
print('Mat std: ', std(mat_canc[2:]))
print('Vec reduction: ', mean(vec_canc[2:]))
print('Vec std: ', std(vec_canc[2:]))
```

13. Generation of Figure 4.5

```
n=200000
o=0
s1 = int(300*fs)
s2 = int(380*fs)
#average the fft of the calibration data to reduce noise
alias_fft,frq_a = avg_fft(liy_cal[:s1],n,o)
#initiate the shape of the aliasing transfer function
A = np.zeros(len(alias_fft),dtype='complex128')
#this is the modulation frequency
w_mod = 1000
#take the Fourier transform of the test data
fft_c = fft.fft(liy_cal[s1:s2])
#Find its corresponding frequencies
frq_c = np.fft.fftfreq(len(fft_c),1/10000)
print(nearest(frq_c,w_mod))
#bin location of the modulation frequency
#  in the calibration data
mod_loc = nearest(frq_a,w_mod)
#implement Equation 2.15 to find the alias transfer function
for i,frq in enumerate(frq_a[:int(len(alias_fft)/2)]):
    if i == mod_loc:
        continue
    A[i] = alias_fft[absolute(i-mod_loc)]/alias_fft[i]
#Interpolate the array so that it can be called easier
A = scipy.interpolate.interp1d(frq_a,A,kind='linear',fill_value='extrapolate')
#Initiate the estimation array
c_alias = np.zeros(len(fft_c),dtype='complex128')
#bin location of the modulation frequency
# in the test data
mod_loc = nearest(frq_c,w_mod)
#Iterate through all frequencies up to 2 times the modulation
#  frequency to determine the estimated aliasing impact
for i in arange(int(2*mod_loc)):
    c_alias[i] = A(frq_c[absolute(mod_loc-i)])*fft_c[absolute(mod_loc-i)]
#Positive and negative frequencies must be complex conjugates for the inverse
#  Fourier transform to result in a real value
c_alias[int(len(c_alias)/2):] = np.flip(conj(c_alias[:int(len(c_alias)/2)]))

fig,ax = subplots(1,3)
figsize(10,5)
ax[0].semilogy(frq_c[:int(len(fft_c)/2)],
               absolute(1e12*fft_c[:int(len(fft_c)/2)]),
               '#1e88e5',linewidth=2)
ax[1].semilogy(frq_c[:int(len(fft_c)/2)],
               absolute(1e12*fft_c[:int(len(fft_c)/2)]),
               '#1e88e5',linewidth=2)
ax[2].semilogy(frq_c[:int(len(fft_c)/2)],
               absolute(1e12*fft_c[:int(len(fft_c)/2)]),
               '#1e88e5',label='Raw',linewidth=2)
```

```
ax[1].semilogy(frq_c[:int(len(fft_c)/2)],
               absolute(1e12*fft_c[:int(len(fft_c)/2)]-
                        1e12*c_alias[:int(len(fft_c)/2)]),
               '#d81b60',linewidth=2)
ax[2].semilogy(frq_c[:int(len(fft_c)/2)],
               absolute(1e12*fft_c[:int(len(fft_c)/2)]-
                        1e12*c_alias[:int(len(fft_c)/2)]),
               '#d81b60',label='Processed',linewidth=2)

ax[1].set_xlabel('Frequency (Hz)',size=15)
ax[0].set_ylabel('LIY (nA)',size=20)
ax[1].tick_params(axis='y',labelleft=False)
ax[2].legend(loc=0,fontsize=13)
ax[2].tick_params(axis='y',labelleft=False)
ax[0].set_xlim(10,110)
ax[1].set_xlim(890,990)
ax[2].set_xlim(1010,1110)
ax[0].set_ylim(1e0,1e6)
ax[1].set_ylim(1e0,1e6)
ax[2].set_ylim(1e0,1e6)
```

14. Implementation of Equations A.3-A.5

```
#Initialize the calibration array of initial current values
I_s,reps = np.zeros(len(step_pnt)),np.zeros(len(step_pnt))
prev = 0
#Iterate through each step within a sweep and
# calculate the initial current values and
# the numebr of points in that step
for idx,point in enumerate(step_pnt):
    I_s[idx] = mean(cur_cut[prev:point])
    reps[idx] = point-prev
    prev = point
reps[-1] = reps[-1] +1
#Create an array of initial current values that are
#  the same shape as the sweep
I_s = repeat(I_s,reps.astype('int'))

#Calibrate the coefficients
C,freqs = calibrate(geo,liy)
I_c = 1e-9 #initial current value set during calibration data collection
#Calculate the Fourier transform of the measured geophone data
#   *This is done over the sweep that the I_s values were found
g_test = geo_cal[:swp_pnt[0]]
liy_test = liy_cal[:swp_pnt[0]]

G_m = fft.fft(g_test)
omega = fft.fftfreq(len(g_test),1/10000) #find the corresponding frequencies
#Interpolate the geophone frequency signal
#  The harmonic calculation does not usually fall onto specific frequency bins,
#  the interpolation estimates the coefficient value if it falls between points
G_m = scipy.interpolate.interp1d(omega,G_m,kind='linear',fill_value='extrapolate')

liy_noise = np.zeros(len(omega),dtype='complex128') #initialize noise array
```

```
#Use the coefficients and interpolated geophone signal to estimate liy_noise
for i in arange(len(C[:])):
    n = i+1
    #Sum the impact that each harmonic has on the signal
    liy_noise += C[i](omega/n)*G_m(omega/n)/
#Find the signal by subtracting noise from measured data (Equation A.5)
liy_sig = liy_test - I_s*fft.ifft(liy_noise)/I_c
```

15. Frequency chirp isolation C.1

```
#the chirp data set is the recorded voltage that was used to drive
#  the speaker, which in turn drove the system. This voltage data set
#  therefore has a very high signal to noise and is used to isolate the
#  beginning and end of the frequency chirp
chirp = dat[:,1]

t = linspace(0,len(chirp)/10000,len(chirp))
start = 100000 #there was a noise at the beginning of the data
end = 300000 #This gives a 20 second window to find the start
             # and a 30 second window to find the end
#Cut the data into beginning and end sections
#   Know the general location of the beginning and end so there
#   is no point in performing the calculations on all the data
chirp_chunk_s = chirp[start:end]
t_chunk_s = t[start:end]
chirp_chunk_e = chirp[-end:]
t_chunk_e = t[-end:]
#Create reference signals for the starting and ending frequencies
# These are the same form as e^(-iwt) detailed in Equation 2.5
wave_s = cos(50*2*pi*t_chunk_s)+1j*sin(50*2*pi*t_chunk_s)
wave_e = cos(400*2*pi*t_chunk_e)+1j*sin(400*2*pi*t_chunk_e)
#Mix the corresponding reference and signal data
lock_s = wave_s*chirp_chunk_s
lock_e = wave_e*chirp_chunk_e
#Put the mixed signals through the corresponding low pass filters
sos_s = signal.butter(6,20,'lowpass',fs=10000,output='sos')
sos_e = signal.butter(6,100,'lowpass',fs=10000,output='sos')
filt_lock_s = signal.sosfiltfilt(sos_s,lock_s)
filt_lock_e = signal.sosfiltfilt(sos_e,lock_e)

#Points 92236 (first freq step) and 216688 (last freq step) were found by
#  adjusting the display parameters to observe when the data stopped/started being
flat
#There are 35000 frequency steps in total ((400-50)/0.01). Therefore, there
#  are 34998 steps between the first step and last step. This is used to find the
#  amount of time between each step and then calculate the beginning and end of
#  the frequency chirp data
x = linspace(t_chunk_s[92236],t_chunk_e[216688],34998)
dx = x[1]-x[0]
start,end = t_chunk_s[92236]-dx,t_chunk_e[216688]+dx
print('dx: ',around(x[1]-x[0],3))
print('Start time:',around(start,3))
print('End time:',around(end,3))

# fig,ax = subplots(2,1)
```

```
fig = plt.figure()
figsize(12,7)
ax,ax1,ax2 =
fig.add_subplot(111,frameon=False),fig.add_subplot(211),fig.add_subplot(212)

ax1.plot(t_chunk_s,filt_lock_s,'#d81b60',linewidth=2,label='50 Hz lockin signal')
ax1.plot(t_chunk_s[92236],filt_lock_s[92236],'k.')
ax1.annotate('First frequency step', xy=(t_chunk_s[92236],filt_lock_s[92236]),
             xytext=(19.8, 0.09),fontsize=18,
             arrowprops=dict(width=3,headwidth=6,headlength=6,facecolor='black',
shrink=0.04)
              )
ax1.legend(loc=1,fontsize=18)
ax1.set_xlim(15,30)
ax1.set_ylim(-0.12,0.12)
ax2.plot(t_chunk_e,filt_lock_e,'#d81b60',linewidth=2,label='400 Hz lockin signal')
ax2.plot(t_chunk_e[216688],filt_lock_e[216688],'k.')
ax2.annotate('Last frequency step', xy=(t_chunk_e[216688], filt_lock_e[216688]),
             xytext=(4222.5, 0.09),fontsize=18,
             arrowprops=dict(width=3,headwidth=6,headlength=6,facecolor='black',
shrink=0.04)
         )
ax2.legend(loc=2,fontsize=18)
ax2.set_xlim(4215,4230)
ax2.set_ylim(-0.12,0.12)
x = linspace(t_chunk_s[92236],t_chunk_e[216688],34998)
dx = x[1]-x[0]
start,end = t_chunk_s[92236]-dx,t_chunk_e[216688]+dx

ax1.annotate('Start of chirp', xy=(start,0.1), xytext=(start, -0.09),
             fontsize=15,arrowprops = {'arrowstyle': '-', 'ls': 'dashed',
                                       'color':'k'},ha='center',va='center',
            )
ax2.annotate('End of chirp', xy=(end,0.1), xytext=(end, -0.09),
             fontsize=15,arrowprops = {'arrowstyle': '-', 'ls': 'dashed',
                                       'color':'k'},ha='center',va='center',
            )
ax.tick_params(labelcolor='none', top=False, bottom=False, left=False,
right=False)
grid(False)
xlabel('Time (secs)',size=20)
ax.set_ylabel('Driving voltage (V)',size=20)
ax.yaxis.labelpad=20
```

16. Sweep change identification C.2

```
dt = t[1] #The change in time during data collection
#Calculate the gradient of the voltage bias
grad = np.gradient(bias,dt)
#Identify the location of the top 20000 gradient values
brk_pnt = np.flip(np.argsort(absolute(grad)))[:2000]
count = 0
#Iterate through all the breakpoints
# Find out how many of the 2000 are valid points
for j in brk_pnt:
```

```
    if absolute(grad[j]) > 1000:
        count = count+1
#Redefine the top count gradient values
brk_pnt = np.flip(np.argsort(absolute(grad)))[:count]
#Order the values by location
brk_pnt = np.flip(np.sort(brk_pnt))
#Iterate through the locations and set any close neighbors to 0
for idx,point in enumerate(brk_pnt):
    temp_pnt = point
    #Set locations in array close to the point to 0
    brk_pnt[np.where(absolute(brk_pnt-point)<10)] = 0
    #This includes the point itself so redefine the point value
    brk_pnt[idx] = point
#Remove any locations that were set to 0
brk_pnt = np.sort(brk_pnt[brk_pnt != 0])

#Figure C.2.1
fig,ax = subplots(2,1)
figsize(15,10)
ax[0].plot(t-15,bias,'#1e88e5',linewidth=2,label='Supplied DC voltage')
ax[0].annotate('Breakpoint', xy=(t[brk_pnt[0]]-15,bias[brk_pnt[0]]),
            xytext=(t[brk_pnt[0]]+0.2-15,bias[brk_pnt[0]]-0.2),
            fontsize=22,arrowprops=dict(facecolor='k', shrink=0.04)
              )
ax[0].annotate('Breakpoint', xy=(t[brk_pnt[1]]-15,bias[brk_pnt[1]]),
            xytext=(t[brk_pnt[1]]+0.1-15,bias[brk_pnt[1]]-0.2),
            fontsize=22,arrowprops=dict(facecolor='k', shrink=0.04)
              )
ax[0].legend(loc=3,fontsize=20)
ax[0].tick_params(labelsize=20)
ax[0].set_xlim(0,2)
ax[0].set_ylim(-0.6,0.6)
ax[0].set_ylabel('Voltage (V)',size=25)


ax[1].plot(t-15,abs(grad),'#d81b60',linewidth=2,label='Voltage gradient')
ax[1].annotate('Largest slope values', xy=(t[brk_pnt[0]]-
            15,abs(grad[brk_pnt[0]])), xytext=(0.5,3000),
            fontsize=22,arrowprops=dict(facecolor='k', shrink=0.04)
              )
ax[1].annotate(' ', xy=(t[brk_pnt[1]]-15,abs(grad[brk_pnt[1]])),
            xytext=(1,3000),
            fontsize=22,arrowprops=dict(facecolor='k', shrink=0.04)
              )
ax[1].legend(loc=3,fontsize=20)
ax[1].tick_params(labelsize=20)
ax[1].set_xlim(0,2)
ax[1].set_ylim(-1e3,6e3)
ax[1].set_ylabel('dV/dt (V/secs)',size=25)
ax[1].set_xlabel('Time (secs)',size=25)


#_____________________________________________________________________________
#Cut the start and end of the data so it aligns with
#   the sweeps properly
start = brk_pnt[0]+3 #The '+3' aligns the start at the very beginning of a sweep
end = brk_pnt[-1]+2 #The '+2' sets the end so it can be identified in future
                    #  gradient calculations
```

```
cur_cut = dat_spec[start:end,0]
geo_cut = dat_spec[start:end,1]
speaker_cut = dat_spec[start:end,2]
bias_mod_cut = dat_spec[start:end,4]
bias_cut = dat_spec[start:end,5]
Z_cut = dat_spec[start:end,6]
LIY_cut = dat_spec[start:end,8]
t = linspace(0,len(cur_cut)/10000,len(cur_cut))

#The locations of the breakpoints need to be redefined
dt = t[1] #This is the change in time
#Calculate the gradient of the cut data
#  Data was cut so that the beginning is the start of a sweep
#  and the end is the end of a sweep
grad = np.gradient(bias_cut,dt)
#identify the top 2000 gradient values
#  these are tied to the breakpoints for reseting the
#  voltage sweep. There are 500 sweeps but multiple
#  points capture the transition from 0.5 V to -0.5 V
swp_pnt = np.flip(np.argsort(absolute(grad)))[:2000]
count = 0
#iterate through all the detected sweep transition points
for j in swp_pnt:
    #Count the number of points above the set threshold
    #  This makes sure that only the points associated with
    #  sweep transition are saved
    if absolute(grad[j]) > 1000:
        count = count+1
#Resort the gradient values and take the top count
swp_pnt = np.flip(np.argsort(absolute(grad)))[:count]
print(count)
#Order the sorted points, there are typically 2-3 neighboring
#  points around the breakpoint that need to be filtered out
swp_pnt = np.flip(np.sort(swp_pnt))
#Iterate through all sorted points, removing close neighbors
print(len(swp_pnt))
for idx,point in enumerate(swp_pnt):
    temp_pnt = point
    #Where a value in the array is close to the point
    # set those values to 0
    swp_pnt[np.where(absolute(swp_pnt-point)<10)] = 0
    #This includes the point itself so redefine the point
    # with its original value
    swp_pnt[idx] = point
#Remove any zeros that were set in the for loop
swp_pnt = np.sort(swp_pnt[swp_pnt != 0])

prev = 0
#iterate through each voltage sweep and determine the
#  location of each step in the sweep
for brk in swp_pnt:
    #Section the grad calculation to the current sweep
    #  *'-1000' ensures the break point between
    #   sweeps is not included
    grad_brk = grad[prev:brk-1000]
    #Determine the top 200 gradient values in this sweep
    #  *In this test data there were only about 100 voltage
```

```
    #    steps. With each step being captured by ~2 points.
    #  **Based on the step magnitude and number this is subject
    #    to change
    pnt = np.flip(np.argsort(absolute(grad_brk)))[:200]

    count = 0
    #Iterate through all 200 gradient values
    for j in pnt:
        #Determine if they meet the expected threshold
        if absolute(grad_brk[j]) > 20:
            count = count+1
    #Redefine the top count gradient values
    pnt = np.flip(np.argsort(absolute(grad_brk)))[:count]
    break

#Iterate through the determined points and remove any neighboring points
for idx,point in enumerate(step_pnt):
    temp_pnt = point
    #Set the value of any location within 10 pins of point to 0
    step_pnt[np.where(absolute(step_pnt-point)<10)] = 0
    #This also sets the point to 0, so reestablish its value
    step_pnt[idx] = point
#Remove any 0 values in the array
step_pnt = np.sort(step_pnt[step_pnt != 0])
#Ensure the last point in the array is indexed
step_pnt = np.append(step_pnt,len(grad_brk)-1)

from mpl_toolkits.axes_grid.inset_locator import (inset_axes, InsetPosition,
                                                  mark_inset)
fig,ax = subplots(2,1)
figsize(15,10)

ax[0].plot(t[:brk-1000],bias_cut[:brk-1000],'#1e88e5',
           linewidth =3,label='Supplied DC voltage')
ax[0].grid(False)
ax[0].tick_params(labelsize=20)
ax[0].set_ylabel('Voltage (V)',size=25)
ax[0].legend(loc=2, fontsize = 25,fancybox=True, framealpha=0)
ax[0].set_xlim(0,1.25)
ax[0].set_ylim(-0.6,0.6)

ax2 = plt.axes([0,0,1,1])
ip = InsetPosition(ax[0], [0.57,0.1,0.4,0.4])
ax2.set_axes_locator(ip)
mark_inset(ax[0], ax2, loc1=2, loc2=3, fc="none", ec='0.5')

ax2.plot(t[:brk-delay],bias_cut[:brk-delay],'#1e88e5',label='Raw',linewidth=3)

ax2.grid(False)
ax2.tick_params(labelsize=12)
ax2.set_xlim(0.45,0.5)
ax2.set_ylim(-0.14,-0.08)

ax[1].plot(t[:brk-1000],abs(grad_brk),'#d81b60',
           linewidth=2,label='Voltage gradient')
ax[1].plot(t[step_pnt],abs(grad_brk[step_pnt-1]),'k.',
          markersize=8,label = 'Step location')
ax[1].tick_params(labelsize=20)
```

```
ax[1].set_xlabel('Time (secs)',size=25)
ax[1].set_ylabel('dV/dt (V/sec)',size=25)
ax[1].legend(loc=2, fontsize = 25,fancybox=True, framealpha=0)
ax[1].set_xlim(0,1.25)
ax[1].set_ylim(-5,80)
```

17. Current sweep cancellation figure

```
from mpl_toolkits.axes_grid.inset_locator import (
    inset_axes, InsetPosition,mark_inset)

prev,count = 0,0
avg_swp = np.zeros((swp_pnt[1]-swp_pnt[0]),dtype='float64')
#Sum through each voltage sweep and sum them together
for brk in swp_pnt:
    avg_swp += cur_cut[prev:brk]
    count += 1
    prev=brk
#Divide by the number of sweeps to get the average
avg_swp = avg_swp/count

sweep = 235 #detail the sweep number to display
delay=10 #a delay is necessary to componsate for time
         #  effects between the geophone/Z and current data
fig,ax1 = subplots()
figsize(10,10)
#Use sweep to determine which sweep to display
loc1 = swp_pnt[sweep-1]
loc2 = swp_pnt[sweep]
ax1.plot(t[:brk-delay],1e9*cur_cut[:brk-delay],
         '#1e88e5',label='Raw')
ax1.plot(t[loc1:loc2-delay],1e9*(
    cur_cut[loc1:loc2-delay]/cur_mat_n[loc1+delay:loc2]),
         '#d81b60',label='Matrix Cancelled')
ax1.plot(t[loc1:loc2-delay],1e9*(
    cur_cut[loc1:loc2-delay]/cur_vec_n[loc1+delay:loc2]),
         '#ffc107',label='Vector Cancelled')
ax1.plot(t[loc1:loc2-delay],1e9*avg_swp[:-delay],
         'b',label='Averaged 500 sweeps',linewidth=2)
ax1.grid(False)
ax1.tick_params(labelsize=20)
ax1.set_xlabel('Time/Voltage (secs/mV)',size=25)
ax1.set_ylabel('Current (nA)',size=25)

ax1.set_xlim(t[loc1],t[loc1]+1.25)
ax1.set_ylim(-1,1.2)
ax2 = plt.axes([0,0,1,1])
ip = InsetPosition(ax1, [1.1,0.1,0.8,0.8])
ax2.set_axes_locator(ip)
mark_inset(ax1, ax2, loc1=2, loc2=3, fc="none", ec='0.5')

ax2.plot(t[loc1:loc2-delay],1e9*cur_cut[loc1:loc2-delay],
         '#1e88e5',label='Raw')
ax2.plot(t[loc1:loc2-delay],1e9*(
    cur_cut[loc1:loc2-delay]/cur_mat_n[loc1+delay:loc2]),
         '#d81b60',label='Matrix Cancelled')
```

```
ax2.plot(t[loc1:loc2-delay],1e9*(
    cur_cut[loc1:loc2-delay]/cur_vec_n[loc1+delay:loc2]),
         '#ffc107',label='Vector Cancelled')
ax2.plot(t[loc1:loc2-delay],1e9*avg_swp[:-delay],
         'b',label='Avg 500 sweeps',linewidth=2)
ax2.grid(False)
ax2.tick_params(labelsize=18)
ax2.set_xlim(t[loc1]+0.9,t[loc1]+1.2)
ax2.set_ylim(0.4,1)
ax2.legend(loc=4, fontsize = 25,fancybox=True, framealpha=0)

#_____________New Cell_____________________________

fig,ax1 = subplots()
figsize(10,8)
#Calculate the Fourier transform of each set of data
fft_cur_swp = 1e12*fft.fft(
    cur_cut[loc1:loc2-delay])/len(cur_cut[loc1:loc2-delay])
fft_cur_mat = 1e12*fft.fft((
    cur_cut[loc1:loc2-delay]/
    cur_mat_n[loc1+delay:loc2]))/len(cur_cut[loc1:loc2-delay])
fft_cur_vec = 1e12*fft.fft((
    cur_cut[loc1:loc2-delay]/
    cur_vec_n[loc1+delay:loc2]))/len(cur_cut[loc1:loc2-delay])
fft_cur_avg = 1e12*fft.fft(avg_swp)/len(avg_swp)

frq_swp = fft.fftfreq(len(fft_cur_swp),1/10000)

ax1.semilogy(frq_swp[:int(len(frq_swp)/2)],absolute(
    fft_cur_swp[:int(len(frq_swp)/2)]),
             '#1e88e5',label='Raw',linewidth=2)
ax1.semilogy(frq_swp[:int(len(frq_swp)/2)],absolute(
    fft_cur_mat[:int(len(frq_swp)/2)]),
             '#d81b60',label='Matrix Cancelled',linewidth=2)
ax1.semilogy(frq_swp[:int(len(frq_swp)/2)],absolute(
    fft_cur_vec[:int(len(frq_swp)/2)]),
             '#ffc107',label='Vector Cancelled',linewidth=2)
ax1.semilogy(frq_swp[:int(len(frq_swp)/2)],absolute(
    fft_cur_avg[:int(len(frq_swp)/2)]),
             'b',label='Avg 500 sweeps',linewidth=2)
ax1.legend(loc=1,fontsize=20)
ax1.tick_params(labelsize=15)
ax1.set_xlabel('Frequency (Hz)',size=20)
ax1.set_ylabel('Current (nA)',size=20)

ax1.set_xlim(50,400)
ax1.set_ylim(1e-1,1e2)
```

18. Synthetic data creation

Functions

```
# Functions used in generating synthetic environmental data
#     create_wave will call the previous functions and create a waveform
#     that incorporates random phase and amplitude changes over time

#designate randomly selected points of time that have a phase slip
```

```
def slipper(t_array,num, debug = False):
    #num: how many phase slips should be incorporated into the frequency
    #indexInt = random.randint(0,int(len(sz)/rate))
    ti = random.choice(arange(len(t_array)),num)
    return ti

#create Phase Map
def phase_map(t_array, loc,debug = False):
    slipSz = t_array.shape
    loc = np.sort(loc)
    PhaseMap = np.ones(slipSz)*(2*np.pi*random.rand(1)-np.pi)
    for point in loc:
        PhaseMap[point:] = 2*np.pi*random.rand(1)-np.pi
    return PhaseMap

#create Amplitude Map
def amp_map(t_array, slip_loc, freq, amp_i=1, mod=0.1, mod_amp=0.05,debug =
False):
    #t_array: the time series values
    #          t_array[1] is used to find the sample rate that was used
    #loc: the index of random time points that a phase shift occurrs
    #freq: frequency that is being modeled. Gaussian width is dependant on freq
    #res_width: resonance width, the bandwidth over
    #               which the power of vibration is greater than
    #               half the power at the resonant frequency
    #gauss_width: width of gaussian curve such that frequency decays to zero at 2
periods
    #amp_i: initial amplitude of function
    #mod: frequency of amplitude madulation
    #mod_amp: amplitude of amplitude modulation
    #debug: for debugging purposes

    period = 1/(freq)
    gauss_width = int(2*period*(1/t_array[1]))

    res_width = 0.0001
    res_freq = 1
    Q_factor = res_freq/res_width

    dropSz = t_array.shape
    slip_loc = np.sort(slip_loc)

    variance = 0.1*amp_i
    t1 =time.time()
    AmpMap = np.ones(dropSz)*amp_i + random.normal(0,variance,1)
    t2 = time.time()
    if debug:
        AmpMap2 = np.copy(AmpMap)
    for point in slip_loc:
        change_up = random.normal(0,variance,1)
        if debug:
            AmpMap2[point:] = 1*amp_i + change_up
        AmpMap[point:] = (1*amp_i + change_up)*exp(
            -(1/Q_factor)*(t_array[point:]-t_array[point]))
        AmpMap[int(point-gauss_width/2):int(point+gauss_width/2)] = 0
    t3 = time.time()

    window = signal.hamming(int(gauss_width))
```

```
    AmpMap = signal.convolve(AmpMap,window, mode='same')/sum(window)
    AmpMap  = AmpMap #+ mod_amp*sin(2*pi*mod*t_array)
    if debug:
        return AmpMap,AmpMap2
    return AmpMap

def noisy(t_array, amp, fs = 10000, cutoff = 1000):
    noise = amp*randn(len(t_array))
    sos = signal.butter(10, cutoff, 'low', fs = fs, output = "sos")
    filtered = signal.sosfilt(sos, noise)
    return filtered

def create_wave(freq_array, amp_array, t_array = None,
                secs = 10, fs = 10000, slips = 10,noise = True):

    if t_array == None:
        t_array = linspace(0,secs,secs*fs)

    if noise:
        low_noise = noisy(t_array,0.000005)
    else:
        low_noise = 0
    tempWave = np.zeros(len(t_array)) + low_noise
    for freq,ampi in zip(freq_array,amp_array):
        ampi = ampi/40000

        ti = slipper(t_array,slips)
        phase = phase_map(t_array,ti)
        amp = amp_map(t_array,ti,freq,amp_i = ampi)
        tempWave += amp*sin(2*np.pi*freq*t_array + phase)

    return tempWave, t_array

#Functions for creating a stepwise frequency chirp
# that does not abruptly just between frequencies
# It drops to 0 with each frequency change over a
#  set number of periods to avoid high frequency noise
def chirp_amp_map(freq_map,t_array):

    period = 1/(freq_map)
    gauss_width = 2*period*(1/t_array[1])

    res_width = 0.0001
    res_freq = 1
    Q_factor = res_freq/res_width

    dropSz = len(t_array)

    loc = np.where(freq_map[:-1] != freq_map[1:])[0]

    gauss_width = gauss_width[loc]
    AmpMap = np.ones(dropSz)

    for point,width in zip(loc,gauss_width):
        width = int(np.around(width))

        point = point + width #width acts as buffer for the
                              #convolution to not affect the ends of the array
```

```
        tempMap = np.ones(int(dropSz+2*width))
        start = int(point-width/2)
        end = int(point+width/2)
        tempMap[start:end] = 0


        window = signal.hamming(width)
        tempMap = signal.convolve(tempMap,window,mode='same')/sum(window)

        AmpMap = AmpMap*tempMap[width:-width]

    return AmpMap,loc

#Time at each frequency is equal to the designated number of periods
#   Output is a waveform of the chirp and the determined time
def chirp_wave(period = 10, res=1,final_freq=300,
               freq_time = 100, timer=False,chirp_type='linear'):

    if timer:
        t1 = time.time()
    final_freq = final_freq+res
    freq_base = linspace(res,final_freq,int((final_freq-res)/res))

    if chirp_type == 'exponential':
        t_len = sum(period/freq_base)
        peat_num = np.around((period/freq_base)*10000).astype('int')
        freq_vec = repeat(freq_base,peat_num)
    if chirp_type == 'linear':
        t_len = (final_freq/res)*freq_time
        peat_num = np.around((t_len/final_freq)*10000).astype('int')
        freq_vec = repeat(freq_base,peat_num)

    t_vec = linspace(0,t_len,t_len*10000)



    vec_dif = int(len(t_vec)-len(freq_vec))

    if vec_dif > 0:   #time vector is bigger take some off end
        t_vec = t_vec[:-vec_dif]
    elif vec_dif < 0:   #freq vector is bigger take some off beginning
        freq_vec = freq_vec[absolute(vec_dif):]
    else:
        pass

    amp,loc = chirp_amp_map(freq_vec,t_vec)

    chirp = amp*np.sin(2*np.pi*freq_vec*t_vec)

    #plot(t_vec,chirp)
    #plot(t_vec,amp)
    # plot(t_vec[loc],chirp[loc],'.')

    if timer:
        print('Chirp generation time: ',np.round(time.time()-t1,2),'sec')

    return chirp,t_vec,loc,freq_base
```

```
def butterworth_lowpass(sig, cutoff, fs = 10000):
    sos = signal.butter(8, cutoff, 'lowpass', fs=fs, output='sos')
    filtered = signal.sosfilt(sos, sig)
    return filtered

def geophone_response(sig, cutoff, fs = 10000, lor_width = 20):
    b,a = signal.butter(sig,cutoff, 'hp', fs = fs)
    w,h = signal.freqz(b,a)
    plt.semilogx(w, 1000*lorentzian(w,cutoff,lor_width)+ 20 * np.log10(abs(h)))
    filtered = 1000*lorentzian(w,cutoff,lor_width)+ 20 * np.log10(abs(h))

def resonant_filter(sig, w_res, res_width, fs = 10000):
    fft_sig = fft.fft(sig)
    fft_freq = fft.fftfreq(len(sig),1/fs)
    sos = signal.butter(8, 12, 'highpass', fs=fs, output='sos')
    filt_freq, filt_response = signal.sosfreqz(sos,worN=len(fft_sig),fs=fs)
    lor_response = lorentzian(fft_freq,w_res,res_width)
    res_response = 1*lor_response+(20*np.log10(absolute(filt_response)))
    res_response[int(len(res_response)/2)+1:] = np.flip(
        res_response[1:int(len(res_response)/2)])
    fft_filt = fft_sig*exp(res_response)
    fft_filt[0] = 0+0j
    return fft.ifft(fft_filt)

def butterworth_highpass(sig, cutoff, fs = 10000):
    sos = signal.butter(10, cutoff, 'hp', fs=fs, output='sos')
    filtered = signal.sosfilt(sos, sig)
    return filtered

def lorentzian(w, w0, g):
    return (1/pi) * (g/2) / ( (w-w0)**2 + (g/2)**2 )

def create_drive(zdata, dt2, tf, freqs):
    fbottom2 = np.fft.fft(zdata)
    freq2 = np.fft.fftfreq(len(zdata),dt2)
    freq2[freq2 > freqs[np.argmax(freqs)]] = freqs[np.argmax(freqs)]
    freq2[freq2 < freqs[np.argmin(freqs)]] = freqs[np.argmin(freqs)]

    driver = np.fft.ifft(tf(freq2)*fbottom2)
    return driver
```

Frequency chirp creation

```
#Create chirp wave, frequency time is determined by number of periods
chirp,t_vce,loc,freq_base = chirp_wave(chirp_type='exponential')
```

Environmental vibration simulation

```
#Simulate frequencies incident to the geophone
#frequencies and magnitudes found upon inspection of data from 2019-02-21
freqs = [1.6, 2, 2.5, 30, 33, 34, 36, 50, 57, 59.5, 100, 101, 111,
         118, 125, 263, 264, 265, 266, 267, 270, 2000,
         2222,2355, 2365,2375,3955,3900,3950,3955,3951,3926,3989]
amps = [0.25, 0.3, 0.28, 0.25,0.3,0.12,0.1,0.1,0.2,0.04,0.035,0.04,0.03,
        0.03, 0.02, 0.02, 0.015, 0.02, 0.01, 0.015, 0.01,0.13,0.16,
        0.12,0.1,0.14,0.1,0.1,0.13,0.11,0.12,0.1,0.1]
start = time.time()

wave,t = create_wave(freqs, amps,secs = 100,slips = 50)
```

Geophone response to vibrations

```
#put data through a high pass filter
# simulate the geophone's response to environmental vibrations
noise_freq = [0.1, 0.5, 1.8, 2.6, 3.1,5.7,6.1,16,25]
noise_amp = [0.25,0.2,0.5,0.425,0.15,0.05,0.1,0.02]
electric_freq = [60,120,180,240,300,360,420,480,540,
                 600,660,720,780,840,900]
electric_amp =  [0.02,0.035,0.2,0.03,0.075,0.025,0.15,0.01,
                 0.04,0.025,0.225,0.2,0.225,0.15,0.03]
# start = time.time()
geo = resonant_filter(wave,30,6)
geo1,t = create_wave(noise_freq, noise_amp,secs = 100,slips = 10)
geo = geo+geo1
# print(time.time()-start)
elec_wave,t = create_wave(electric_freq,
                          electric_amp,secs = 100,slips=0,noise=False)
geo = geo + elec_wave
fft_geo = np.fft.fft(geo)
freq_geo = np.fft.fftfreq(len(geo),1/10000)
semilogy(freq_geo[:int(len(fft_geo)/2)],abs(fft_geo[:int(len(fft_geo)/2)]/100))
xlim(0,800)
ylim(1e-5,0.5)
title('Synthetic Geophone Frequency Response')
```

Transfer function to tip idealization

```
#load real, calibrated transfer function
[transf_Alb, freqs_Alb] = joblib.load('Albert_Data/2019-02-26/trx2_v2')

transf = transf_Alb
freqs = freqs_Alb
lnfrq = int(len(freqs)/2)

# print(freqs[1])
#create log spaced
w_log = logspace(log10(freqs[1]),log10(freqs[-1+int(len(freqs)/2)]),100000)

T_log = transf(w_log) #transf is already an interpolation of frequencies

window = signal.hamming(int(300))
T_flt = signal.convolve(T_log,window, mode='same')/sum(window)

# plot(w_log,absolute(T_log))
# plot(w_log,absolute(T_flt))


f = scipy.interpolate.interp1d(w_log,T_flt, kind='linear')

w_lin = linspace(w_log[0],w_log[-1],lnfrq-1)
transf_flt = np.zeros(shape(freqs),dtype=np.complex128)

transf_flt[1:lnfrq] = f(w_lin)
transf_flt[lnfrq:] = np.flip(transf_flt[:lnfrq])

interp_transf = scipy.interpolate.interp1d(freqs,transf_flt, kind='linear')
phase = angle(transf(freqs))
# plot(freqs[:lnfrq],phase[:lnfrq])
plot(freqs[:lnfrq],absolute(transf_flt[:lnfrq]))
```

```
# plot(freqs,abs(transf_flt))
xlim(0,200)
```

Tip harmonic response to geophone

```
def chirped_tip(chirp_wave, geo_wave, t_array, freq_val,
wave_loc,transfer_function,
                harmony = 'square', chirp_type = 'linear',num = 5):
    #input is chirp and geophone data seperately

#=============================================================================================
======================
    #======================Add harmonics to the existing waveform
=========================================

#=============================================================================================
======================
    wave = chirp_wave+geo_wave

    if harmony == 'square':
        harmonics = arange(int(num-1))*2+3
        amps = 1/harmonics
    i=1
    prev_point = 1000000
    for val_pnt,loc_pnt in zip(freq_val,wave_loc):
        temp_t = np.copy(t_array)
        chirp_part = np.zeros(len(wave))
        if i == 0:
            chirp_part[0:loc_pnt]= chirp_wave[0:loc_pnt]
            temp_t[loc_pnt:] = 0
            i=1
        else:
            chirp_part[prev_point:loc_pnt] = chirp_wave[prev_point:loc_pnt]
            temp_t[loc_pnt:] = 0
            temp_t[:prev_point]=0
        harm_freq = harmonics*val_pnt
        chirp_mag = amps*max(chirp_part)
#         plot(t_array,chirp_part)
        for freq,mag in zip(harm_freq,chirp_mag):
            wave = wave + mag*np.sin(freq*2*np.pi*temp_t)
#             plot(t_array,wave)
        prev_point=loc_pnt

#=============================================================================
#====Output of the original waveform plus added harmonics of chirp frequencies===

#=============================================================================
#============Apply the standard transfer function to resulting wave=======
    freqs = np.fft.fftfreq(len(wave),1/10000)
    chirp_tip = create_drive(wave,1/10000,transfer_function,freqs)

    return wave,chirp_tip
```

Budget:

This budget reports any cost incurred during this ES100 project and does not reflect purchases made by the Hoffman lab for equipment or purchases made in the past ES100 project.

Purchases	Cost
None	$0

Works Cited

[1] M. Jacoby, "30 years of moving atoms: How scanning probe microscopes revolutionized nanoscience," C&EN, 19 November 2019. [Online]. Available: https://cen.acs.org/analytical-chemistry/imaging/30-years-moving-atoms-scanning/97/i44. [Accessed 9 March 2020].

[2] M. F. Crommie, "Confinement of Electrons to Quantum Corrals on a Metal Surface," *Science,* vol. 262, no. 5131, pp. 218-220, 1993.

[3] "Scanning Tunneling Microscopy (STM)," Park Systems, [Online]. Available: https://parksystems.com/park-spm-modes/94-electrical-properties/241-scanning-tunneling-microscopy-stm. [Accessed 8 4 2020].

[4] "Scanning Tunneling Microscopy," nanoScience Instruments, [Online]. Available: https://www.nanoscience.com/techniques/scanning-tunneling-microscopy/. [Accessed October 2019].

[5] "STM Measurement Types," Hoffman Lab, 1 January 2010. [Online]. Available: http://hoffman.physics.harvard.edu/research/STMmeas.php. [Accessed September 2019].

[6] M. Hamidian, "Private Communication," 2016.

[7] A. Chen, "Vibration Mitigation for Atomic-Resolution Imaging," 5 April 2019. [Online]. Available: http://hoffman.physics.harvard.edu/publications/theses/2019-04-15-Albert-Chien-BSc-thesis.pdf.

[8] B. Primavera, "Cancelling picometer vibrations from a scanning tunneling microscope by post processing," in *American Physical Society*, https://meetings.aps.org/Meeting/MAR18/Session/A01.2, Los Angeles, 2018.

[9] H. Pirie, "Private Communication," 2017.

[10] Oliva et al., "Vibration isolation analysis for a scanning tunneling microscope," *Review of Scientific Instruments,* vol. 63, no. 6, pp. 3326-3329, 1992.

[11] Iwaya et al., "Systematic analyses of vibration noise of a vibration isolation system for high-resolution scanning tunneling microscopes," *Review of Scientific Instruments,* vol. 82, no. 083702, 2011.

[12] Liu et al., "Active mechanical noise cancellation scanning tunneling microscope," *Review of Scientific Instruments,* vol. 78, no. 073705, 2007.

[13] Pabbi et al., "ANITA - An active vibration cancellation system for scanning probe microscopy," *Review of Scientific Instruments,* vol. 89, no. 063703, 2018.

[14] W. Flynn, "What is Total Harmonic Distortion (THD)?," Siemens PLM Software, 18 September 2020. [Online]. Available: https://community.sw.siemens.com/s/article/what-is-total-harmonic-distortion-thd. [Accessed 11 January 2020].

[15] "Principles of Lock-In Detection," Zurich Instruments, [Online]. Available: https://www.zhinst.com/americas/resources/principles-lock-detection. [Accessed 29 January 2020].